Computer Persönlich

Ein Markt & Technik Buch

Der Einstieg in FORTH

Die Programmiersprache FORTH im Selbststudium erlernen.
Mit zahlreichen Übungsaufgaben.

Paul M. Chirlian

Paul M. Chirlian

Der Einstieg in FORTH

Die Programmiersprache FORTH im Selbststudium erlernen. Mit zahlreichen Übungsaufgaben.

Deutsche Übersetzung
Peter Rosenbeck

Markt & Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Chirlian, Paul M.:
Der Einstieg in FORTH : d. Programmiersprache FORTH im Selbststudium erlernen ;
mit zahlr. Übungsaufgaben / Paul M. Chirlian.
Dt. Übers.: Peter Rosenbeck. — Haar bei München : Markt-und-Technik-Verlag, 1985.—
(Computer persönlich)
Einheitssacht.: Beginning FORTH <dt.>
ISBN 3-89090-085-2

Titel der amerikanischen Originalausgabe
»Beginning FORTH« by Paul M. Chirlian
ISBN 0-916460-36-3

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

89 88 87 86 85

ISBN 3-89090-085-2

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Eimannsberger, München
Printed in Germany

Inhaltsverzeichnis

1

Einführung in das Programmieren mit FORTH

1 Einführung in das Programmieren mit FORTH

In diesem Buch geht es um die Programmiersprache FORTH und darum, wie man in dieser äußerst vielseitigen Sprache Programme schreiben kann. Wir machen Sie zuerst mit den grundlegenden Konzepten der Programmierung vertraut, die Sie dann zu sehr leistungsfähigen komplexen Prozeduren auszubauen lernen.

Die modernen elektronischen Computer verfügen über erstaunliche Fähigkeiten. In ein paar Sekunden oder Minuten können sie Aufgaben erledigen, deren manuelle Bewältigung ansonsten Monate oder Jahre in Anspruch nehmen würde. Computer steuern industrielle Fertigungsprozesse oder überwachen medizinische Geräte in Notfallstationen. Dennoch: Der Computer kann nicht denken! Er kann lediglich eine Folge von genauen Anweisungen befolgen, die den Namen Programm trägt. In diesem Buch stellen wir eine Sprache - genannt FORTH - dar, mit der man Computer programmieren kann. Im ersten Kapitel befassen wir uns dazu mit ein paar grundlegenden Fakten über Computer und vergleichen außerdem FORTH mit anderen Programmiersprachen. Das Kapitel unternimmt aber auch bereits die ersten "Programmierschritte" mit Ihnen, so daß Sie bereits Ihr erstes Programm schreiben können.

1.1 Grundlegendes über Computer

Ehe wir uns dem Programmieren in FORTH zuwenden können, sollten wir uns in groben Zügen mit dem Aufbau eines Computers vertraut machen. Zwar beziehen sich unsere Ausführungen hauptsächlich auf Mikro- oder Personal Computer (das sind Computer, die auf einem sog. "Mikroprozessor" basieren), im Prinzip entspricht aber deren Aufbau auch dem der wesentlich teureren (und leistungsstärkeren) Großcomputer.

Das Herzstück eines Mikrocomputers ist die sog. CPU (Abkürzung für "central processing unit", wörtlich "Zentrale Verarbeitungseinheit" oder "Zentraleinheit"). Die CPU führt alle logischen und arithmetischen Operationen aus, die der Computer beherrscht. Sie kann z.B. zwei Zahlen zueinander addieren oder sie daraufhin untersuchen, ob sie gleich sind. Außerdem verfügen Computer über einen Hauptspeicher; darin werden die Programme gespeichert, die

die Funktionsweise des Rechners steuern, aber auch die Daten, mit denen diese Programme arbeiten.

Der Hauptspeicher kann die CPU äußerst schnell mit Daten versorgen, oftmals in Bruchteilen einer Mikrosekunde (Millionstelsekunde). Diese Art von Speicher ist jedoch verhältnismäßig teuer; außerdem können die meisten Mikrocomputer nur mit einem begrenzten Umfang an Hauptspeicher arbeiten (bei 8-Bit-Mikrocomputern sind das in der Regel Speichergrößen von 64K). Deshalb kann man den Hauptspeicher nicht zur Speicherung größerer Datenmengen benutzen. Es kann z.B. sein, daß Sie über mehrere Programme verfügen, die Sie bei verschiedenen Gelegenheiten einsetzen wollen. Es wäre nun äußerst unökonomisch, all diese Programme stets im Hauptspeicher aufzubewahren, da dieser dadurch sehr bald voll wäre. Außerdem müßten Sie Ihren Computer stets eingeschaltet lassen, denn der Inhalt des Hauptspeichers geht verloren, wenn keine Spannung mehr anliegt. (Einige Computer verfügen über Speicher, die Informationen auch dann noch behalten können, wenn der Computer ausgeschaltet ist. Leider ist diese nützliche Eigenheit bei den meisten Mikrocomputern nicht gegeben.)

Ähnlich verhält es sich mit der Speicherung von Daten. Stellen Sie sich einmal eine Bank vor, die über 100000 Konten zu führen hat. Der Computer dieser Bank müßte über einen gigantischen Hauptspeicher verfügen, um alle Informationen über Kontobewegungen speichern zu können. Schon aus finanziellen Gründen scheidet also diese Lösung aus. Deshalb bedient man sich bei der Speicherung von Computerdaten magnetischer Speichermedien. Es ist möglich, auf <u>Magnetband</u> eine große Menge von Informationen zu speichern. Bei kleineren Computern findet man für diesen Zweck oft einen gewöhnlichen Kassettenrecorder. Ein einziges Band (bzw. Kassette) kann eine Menge von Programmen aufnehmen; wenn Sie eines dieser Programme laufen lassen wollen, dann lesen Sie es erst vom Band in den Hauptspeicher des Computers ein, von wo aus es der Computer dann ausführt. Andere Speichermedien, die man zusammen mit Mikrocomputern antrifft, sind <u>Disketten</u> oder <u>Floppy disks</u> und <u>Festplatten</u> (<u>Hard disks</u>). Sie haben vor Kassettenspeichern den Vorteil eines wesentlich schnelleren Datenzugriffs. In letzter Zeit sind auch die sog. <u>Magnetblasenspeicher</u> für Mikrocomputer verfügbar geworden.

Um überhaupt mit dem Computer etwas anfangen zu können, müssen wir in der Lage sein, ihm Programme und Daten einzugeben und uns

die Ergebnisse seiner Berechnungen irgendwie anzusehen. Bei den meisten kleinen Computern dient ein Terminal für die Ein- und Ausgabe von Informationen. Zur Eingabe von Informationen bedient sich der Benutzer einer Tastatur, die der einer Schreibmaschine ähnelt. Die Ausgaben des Computers erscheinen auf dem Bildschirm, ähnlich einem gewöhnlichen Fernsehbildschirm (bei einigen der billigeren Mikrocomputern handelt es sich sogar um einen Fernseher). Manchmal ist auch noch ein Drucker an den Mikrocomputer angeschlossen, so daß man sich die Ergebnisse von Berechnungen auch ausdrucken lassen kann (die Fachleute sprechen in diesem Zusammenhang von "Hardcopy").

1.2 Programmiersprachen

Alle Informationen werden im Computer in Form von Zahlen gespeichert. Diese Zahlen bestehen lediglich aus den Ziffern 0 und 1; man nennt sie auch Binärzahlen. So ist z.B. die Folge 0110 eine Binärzahl, die für die gewöhnliche (Dezimal-) Zahl 6 steht. Sie müssen nichts über Binärzahlen wissen, um einen Computer in FORTH zu programmieren. Einer der Gründe für den Einsatz einer Programmiersprache wie FORTH ist es ja gerade, daß man dadurch mit dem Computer nicht mehr in der "Sprache" der Binärzahlen zu reden braucht.

Man kann zur Programmierung eines Computers auch die sogenannte Maschinensprache verwenden. Die Befehle in dieser Sprache sind verhältnismäßig einfach; sie weisen z.B. den Computer an, zwei Zahlen zu addieren oder eine Zahl an einer bestimmten Stelle im Hauptspeicher abzuspeichern. Befehle in Maschinensprache bestehen aus langen Folgen von Nullen und Einsen. Deshalb ist die Arbeit mit diesen Sprachen äußerst umständlich und fehleranfällig. Schon eine einfache Multiplikationsaufgabe erfordert, daß der Programmierer eine lange Folge von Maschinenbefehlen (die nur aus den Ziffern 0 und 1 bestehen dürfen) hinschreibt. Um den Programmierern das Leben zu erleichtern, hat man deshalb besondere Programme entwickelt, die einfachere und leichter zu verstehende Befehle (z.B. in Englisch) in die Sprache der Nullen und Einsen überführen, die der Computer versteht.

Es gibt eine ganze Menge solcher Programme, da es ja auch eine ganze Menge Programmiersprachen gibt. Dennoch lassen sich dabei

zwei Klassen unterscheiden. Da sind zum einen die sog. Assembler. Assemblerprogramme verstehen nur Befehle, die sehr eng an die Maschinensprache angelehnt sind, für die Befehle jedoch nicht mehr ausschließlich 0 und 1, sondern sog. symbolische Ausdrücke verwenden. Die Sprache, die ein solches Assemblerprogramm versteht, heißt auch einfach Assembler. Bei der Arbeit in Maschinensprache muß der Programmierer nicht nur ausschließlich Binärzahlen eingeben, er muß sich auch um die tatsächlichen Speicheradressen kümmern, an denen seine Programme stehen. Auch diese Arbeit erleichtert ihm der Assembler. Hier kann man sich mit Hilfe von symbolischen Ausdrücken (der sog. "Marken") auf Speicheradressen beziehen, ohne daß man weiß, wo genau sich diese Speicheradresse befindet. Trotz dieser Erleichterungen ist die Assemblersprache dennoch nichts anderes als eine etwas leichter zu handhabende Form von Maschinensprache.

Die andere Klasse von Programmiersprachen, die wir oben erwähnt haben, sind die sog. höheren Programmiersprachen. Diese erleichtern die Arbeit des Programmierers beträchtlich, da sie Befehle in eine ganze Folge von Maschineninstruktionen übersetzen. So verfügen die meisten höheren Programmiersprachen über das Symbol "*", um damit die Multiplikation auszudrücken. Der Computer kennt jedoch den Stern als Befehl nicht. Deshalb wird ein Stern in eine Folge (oftmals 100 und mehr) von Maschinenbefehlen übersetzt, die dafür sorgen, daß der Computer multipliziert. Mit den Details der Übersetzung des Sterns braucht sich der Programmierer nicht zu befassen. Um alle diesen lästigen Kleinkram kümmert sich die Programmiersprache selbst.

Höhere Programmiersprachen sind also bequemer als Maschinen- oder Assemblersprachen, dafür sind sie aber auch etwas eingeschränkter. Bei der Arbeit in Assembler oder Maschinensprache können Sie jeden der eingebauten Befehle einsetzen, die Ihr Computer versteht. Sie haben dadurch die Möglichkeit, bei genauer Kenntnis der Arbeitsweise Ihres Computers möglichst viel aus ihm herauszuholen. Arbeiten Sie aber in einer höheren Programmiersprache, dann müssen Sie sich mit den Befehlen zufriedengeben, die darin enthalten sind. Die meisten höheren Programmiersprachen sind jedoch schon ziemlich flexibel und verfügen über ein umfangreiches Befehlsrepertoire; außerdem kann man mit ihnen unendlich viel leichter umgehen als mit Assembler oder Maschinensprache. In diesem Buch stellen wir Ihnen eine Programmiersprache vor, die noch flexibler und "ausdrucksreicher" als die herkömmlichen höheren Programmiersprachen ist, nämlich FORTH!

1.3 FORTH und andere Programmiersprachen

FORTH ist nur eine von vielen Programmiersprachen, die auf Mikrocomputern eingesetzt werden. Andere weit verbreitete Sprachen sind BASIC, FORTRAN, Pascal und, in etwas geringerem Umfang, COBOL. FORTH unterscheidet sich ganz wesentlich von all diesen genannten Sprachen. Am deutlichsten sieht man dies an einem Beispiel: Schreiben wir einmal ein Programm in BASIC und FORTH, das die beiden Zahlen 3 und 4 addiert und das Ergebnis (7) ausdruckt. Da wir davon ausgehen, daß Sie noch über keine Programmiererfahrung verfügen, haben wir uns auf so ein simples Beispiel beschränkt. Das BASIC-Programm sieht folgendermaßen aus:

```
10 A = 3
20 B = 4
30 C = A + B
40 PRINT C                                        (1-1)
```

Hier nun das FORTH-Programm:

```
3 4 + .                                           (1-2)
```

Schon an diesem einfachen Beispiel kann man einige Vorteile (und Nachteile) von FORTH erkennen. Das BASIC-Programm sieht ziemlich vertraut aus, da es ein bißchen an die Schularithmetik erinnert, während das FORTH-Programm exotisch anmutet. Unbestritten ist das FORTH- Programm jedoch wesentlich kürzer. Wenn Sie lange Programme schreiben wollen, dann werden Sie diese Knappheit zu schätzen wissen, da sie nicht soviel Schreibarbeit haben. Weil sie nicht so lang sind, benötigen FORTH-Programme aber auch weniger Speicherplatz und laufen auch in den meisten Fällen schneller. Andererseits sind lange Programme oft "selbstdokumentierend". Wenn sie sauber geschrieben sind, dann erklären sich diese Programme selbst, und man kann durch bloßes Durchlesen dem Programm entnehmen, wozu es dient und wie es seine Aufgabe erledigt. Wir werden aber sehen, daß man in ein FORTH-Programm Kommentare aufnehmen und auch sie dadurch selbstdokumentierend machen kann.

1 Einführung in das Programmieren mit FORTH

Im letzten Abschnitt haben wir kurz erwähnt, daß Maschinen- oder Assemblersprachen flexibler als höhere Programmiersprachen sind. FORTH verfügt nun über einige Aspekte, die den Assembler- oder Maschinensprachen entsprechen und hat deshalb auch viel von ihrer Flexibilität. Diese Flexibilität erkauft man sich jedoch nicht auf Kosten der Handhabung.

Bevor wir einige weitere Gesichtspunkte von FORTH erörtern können, müssen wir uns mit einigen anderen wichtigen Prinzipien höherer Programmiersprachen vertraut machen. Meistens wird ein Programm in einer höheren Programmiersprache durch einen sog. Compiler in Maschinensprache übersetzt. Diesen Übersetzungsprozeß bezeichnet man deshalb auch als Compilierung. Nachdem wir das Programm geschrieben haben, übergeben wir es dem Compiler, und der Prozeß der Compilierung beginnt. Das Ergebnis des Compilers ist aber immmer noch kein lauffähiges Programm. Meistens gibt es noch eine Anzahl von Unterprogrammen, die erst mit dem eigentlichen Programm kombiniert werden müssen, um es zu vervollständigen. So könnte Ihr Programm etwa eine Anzahl von Multiplikationen enthalten. Der Compiler schreibt nun nicht eine lange Folge von Maschineninstruktionen für jede Multiplikation in Ihrem Programm. Statt dessen weiß er, daß in einer sog. Bibliothek (Fachausdruck: "Library") ein Unterprogramm zu finden ist, das sich um die Multiplikation kümmert. Diese Programmbibliotheken werden zusammen mit dem Compiler vom Hersteller ausgeliefert. Nach der Compilierung befinden sich in Ihrem Programm Befehle, die das Multiplikationsprogramm aus der Bibliothek aufrufen. Um das Programm zu vervollständigen, muß also erst noch das Multiplikationsprogramm aus der Bibliothek geholt und mit dem selbst geschriebenen Programm kombiniert werden. Diesen Prozeß bezeichnet man als Binden (Fachausdruck: Linken).

Compilieren und Linken nimmt beträchtliche Zeit in Anspruch. Auf kleineren Computern kann es schon passieren, daß das Compilieren und Linken eines mittleren Programms fünf Minuten und länger dauert, wenn Sie z.B. mit einer Sprache wie FORTRAN arbeiten. Erschwerend kommt hinzu, daß neue Programme meistens einige Fehler enthalten (Diese Fehler werden im Programmiererjargon Bugs, vom englischen "bug" = Wanze genannt.) Wenn Bugs in Ihrem Programm sind, müssen Sie Ihr Programm verbessern und es erneut compilieren und linken lassen. Da leider die meisten Programme irgendwelche Bugs enthalten, geht einem diese fünfminütige Warterei ziemlich schnell auf die Nerven! Sie können den ganzen Prozeß zwar etwas abkürzen, indem Sie Ihr Programm in mehrere kleine Unter-

Programme aufteilen. Diese Programmteile oder *Moduln* können schnell compiliert und, wenn sie fehlerfrei sind, in die Programmbibliothek mitaufgenommen werden. Bei dieser Art Programmentwicklung muß immer nur das zuletzt entwickelte Modul compiliert und getestet werden, wodurch sich die Compilierzeit verringert. Immer noch müssen Sie aber zur Erstellung eines lauffähigen Programms die einzelnen Unterprogramme mit Hilfe des Linkers aus der Bibliothek holen lassen. Deshalb ergeben sich nach wie vor bei jedem Entwicklungsschritt mehrere Minuten Wartezeit. Einer der großen Vorteile von FORTH besteht nun darin, daß jedes Programm in sehr kleine Einzelschritte zerlegt werden kann, so daß die Compilationszeiten reduziert werden. Bedeutsamer aber ist, daß in FORTH *der Prozeß des Linkens völlig wegfällt*. Dadurch läßt sich die Entwicklung eines Programms in FORTH wesentlich beschleunigen.

Bei einigen Programmiersprachen wie z.B. BASIC ist das Problem der Wartezeiten bei der Programmentwicklung anders gelöst; diese Sprachen werden nicht compiliert, sondern *interpretiert*. Man spricht deshalb in diesem Zusammenhang auch von einem *Interpreter*. Hier läuft der Programmierprozeß etwas anders ab. Wenn man ein Programm compiliert und linkt, dann erhält man als Ergebnis ein Maschinenprogramm. Sie lassen dieses Programm laufen, um die gewünschten Daten zu erhalten. In einer interpretierten Sprache erhalten Sie niemals ein Maschinenprogramm. Statt dessen liest ein anderes Programm, eben der Interpreter, Ihre Befehle durch und stößt entsprechende Sequenzen von Maschinenbefehlen an, die das bewirken, was Sie haben wollen. Danach liest der Interpreter den nächsten von Ihnen eingegebenen Befehl und führt ihn aus usw. Interpretersprachen führen Befehle also sofort aus, nachdem sie eingegeben sind. So geht keine Zeit für Compilieren und Linken verloren.

Der Nachteil dieses Verfahrens liegt darin, daß interpretierte Programme nur sehr langsam laufen, oft zehn- bis dreißigmal langsamer als ein compiliertes Programm. In einigen Fällen ist die Aufgabenstellung für den Computer so einfach, daß dies für den Benutzer keinen Unterschied ausmacht. Ob er nun 0,01 oder 0,2 Sekunden wartet, ist ihm wohl egal. Andererseits ist der Unterschied zwischen einer Wartezeit von 10 Minuten und 30 Sekunden schon gravierend. Deshalb können interpretierte Programme problematisch werden. BASIC bietet hier einen Ausweg: Man kann das Programm im Interpreter entwickeln und testen; sobald es fertig ist, übergibt man es einem speziellen BASIC-Compiler, der es in

eine schnellere compilierte Version überführt. Oft ist dies das bestmögliche Vorgehen. Die langsame Laufzeit der interpretierten BASIC-Programmme kann aber wiederum den Prozeß der Fehlersuche unerträglich verzögern. Einer der großen Schwachpunkte des interpretierten BASIC ist nämlich, daß der Programmierer keine Unterprogrammme oder Moduln schreiben kann, wodurch eine modulare Programmentwicklung nicht mehr möglich ist. Stets muß für die Fehlersuche das gesamte Programm ablaufen. Dennoch kann man sagen, daß der Vorteil interpretierter Sprachen darin liegt, daß Fehler sehr schnell lokalisiert und ausgebessert werden können, da die Programme sofort nach dem Eingeben ausgeführt werden.

Die Sprache FORTH zeichnet sich dadurch aus, daß sie sowohl Merkmale von Assemblersprachen als auch solche von höheren Sprachen aufweist. Wenn wir die Maschinensprache als die erste Ebene der Programmiersprache auffassen und höhere Sprachen sich auf der zweiten bzw. dritten Ebene befinden, dann ist FORTH eine Sprache der vierten Ebene. Daher kommt übrigens auch der Name FORTH. Es wurde von Charles H. Moore bei der Arbeit mit einem Computer der dritten Generation entwickelt. Seine neue Sprache war so mächtig, daß sie aus seinem Rechner einen Rechner der vierten Generation zu machen schien. Deshalb wollte Moore ihr den Namen FOURTH geben (vom Englischen "fourth": viertens oder Viertes). Leider akzeptierte sein Computer aber keine Befehle, die mehr als fünf Buchstaben enthielten; so entstand der Name FORTH.

Wenn wir FORTH mit anderen Programmiersprachen vergleichen, so sehen wir, daß es für den Anfänger etwas schwieriger zu erlernen ist, und daß die Programme nicht unbedingt selbstdokumentierend sind. Andererseits kann man in FORTH aber sehr kurze Programme schreiben, deren Fehler man schnell entdeckt und korrigiert hat, da auch die Compilierzeit eines FORTH-Programms im Verhältnis zu anderen Programmiersprachen wesentlich reduziert ist. Die Laufzeit von FORTH-Programmen ist wesentlich geringer als die entsprechender interpretierter BASIC-Programme. Sie kann zwar etwas länger als die eines compilierten FORTRAN-Programms sein, dafür verkürzt sich aber die Compilier- und Linkzeit in FORTH. Einer der Hauptvorteile von FORTH ist außerdem, daß die Sprache erweiterbar ist: der Programmierer kann die Sprache selbst um neue Befehle erweitern. Darauf werden wir später noch eingehen. Ein weiterer Vorteil von FORTH im Vergleich mit anderen Programmiersprachen ist es, daß FORTH wesentlich weniger Arbeitsspeicher benötigt. Gerade beim Einsatz mit kleinen Computern erweist sich dies als unschätzbarer Vorteil.

Bei der Arbeit mit Großrechnern sind die Vorteile von FORTH nicht so unmittelbar deutlich. Diese Computer sind meist sehr schnell und verfügen über eine enorme Menge an Hauptspeicher. Die Zeit für die Compilierung und das Linken eines Programms kann so klein sein, daß sie für den Programmierer gar nicht mehr ins Gewicht fällt. Ebenso zählt bei Großrechnern der verminderte Speicherplatzbedarf nicht so sehr. Es verbleibt aber immmer noch die Erweiterbarkeit von FORTH als ein großer Vorteil.

1.4 Zum Anfang: Der "Stack" - die Postfix-Schreibweise

Wir wenden uns nun den grundlegenden Ideen von FORTH zu, damit Sie Ihr erstes Programm schreiben können. Als erstes wollen wir noch einmal ein einfaches Programm schreiben, das die Zahlen 3 und 4 addiert und das Ergebnis der Addition, 7, ausdruckt. In FORTH sieht das so aus:

```
3 4 + . (RETURN)                                        (1-3)
```

Mit (RETURN) wollen wir ausdrücken, daß Sie die Return-Taste auf Ihrem Terminal betätigen sollen (und nicht etwa das Wort "return" einschreiben). Sehen wir uns nochmal etwas genauer an, wie man dieses Programm eingibt. Erst tippen wir die 3, gefolgt von einem Leerzeichen, dann die 4 und wieder ein Leerzeichen. Anschließend geben wir ein Pluszeichen ein, gefolgt von einem Leerzeichen, auf das ein Punkt folgt. Nachdem wir all dies eingegeben haben, drücken wir die Return-Taste. Der Computer antwortert darauf mit

```
7 ok                                                    (1-4)
```

Beachten Sie die Leerzeichen. Die meisten Programmiersprachen ignorieren Leerzeichen. Anders bei FORTH: Hier dienen die Leerzeichen als Trennzeichen, d.h., sie dienen dazu, einzelne Befehle voneinander zu trennen. Deshalb merken Sie sich: Leerzeichen sind in FORTH sehr wichtig!

Natürlich müssen Sie, ehe Sie dieses Programm eingeben können, Ihr FORTH-System erst in den Computer bringen oder "laden". Wie Sie das machen, entnehmen Sie am besten Ihrem FORTH-Handbuch.

Betrachten wir nun das Beispielprogramm (1-3) etwas genauer. Zuerst geben wir dem Computer die Zahlen 3 und 4 als Daten ein. Der Computer merkt sich diese Daten in einem ganz bestimmten Bereich des Arbeits- (oder Haupt-)speichers, dem sog. Stack. Als nächstes trifft der Rechner auf das Pluszeichen. Dies stellt einen Befehl dar, der ihn dazu veranlaßt, die beiden Werte 3 und 4 zu addieren, die soeben auf dem Stack gespeichert wurden. Dabei werden die 3 und die 4 vom Stack entfernt und statt dessen ihre Summe 7 dort abgelegt. Schließlich stößt der Computer auf den Befehl (!)".". Am Anfang ist es etwas seltsam, daß ein Interpunktionszeichen in einer Programmiersprache einen eigenen Befehl darstellt. In FORTH sorgt dieser Befehl aber dafür, daß das Element, das sich gerade zuoberst im Stack befindet, auf dem Bildschirm (oder Drucker) ausgegeben wird. Wir werden diese Dinge noch eingehender darstellen.

Sehen wir uns dazu den Stack einmal genauer an. Der Stack ist nichts anderes als eine Folge von Speicherstellen im Hauptspeicher des Computers. Eine bildliche Darstellung des Stacks sehen Sie in Abbildung 1-1. In 1-1a ist der Stack noch leer: es sind bisher noch keine Informationen in den Rechner eingegeben und auf dem Stack gespeichert worden.

Nehmen wir jetzt einmal an, wir lassen das Programm (1-3) ausführen, weisen also den Computer an, die darin enthaltenen Befehle zu befolgen. Dazu "liest" sich der Computer das eingegebene Programm von links nach rechts durch. Er trifft als erstes auf die Ziffer 3. Er weiß, daß dies ein Datenelement ist, das er auf dem Stack speichern muß. Man sagt in diesem Zusammenhang im Programmiererjargon, daß die 3 auf den Stack gepusht wird. Wie dies aussieht, können Sie der Abbildung 1-1b entnehmen; hier ist die 3 das oberste Element auf dem Stack. Jetzt aber weiter mit Programm (1-3). Der Rechner trifft als nächstes auf die 4. Auch dies ist ein Datenelement und muß deshalb auf den Stack gepusht werden. Wir erhalten die Abbildung 1-1c. Beachten Sie, daß sich die Drei jetzt um einen Eintrag weiter "unten" befindet und die Vier das neue oberste Element auf dem Stack ist. Jedesmal, wenn nämlich Daten auf den Stack gepusht werden, wandern alle bereits vorhandenen Einträge um eine Position nach unten, und das neue Datenelement wird zum neuen obersten Eintrag.

```
I----I     I----I     I----I     I----I     I----I
I    I     I 3  I     I 4  I     I 7  I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I 3  I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I    I     I    I     I    I     I    I     I    I
I----I     I----I     I----I     I----I     I----I
I .  I     I .  I     I .  I     I .  I     I .  I
I .  I     I .  I     I .  I     I .  I     I .  I
  .          .          .          .          .

  a)         b)         c)         d)         e)
```

ABBILDUNG 1-1: Ein Beispiel für die Stack-Operationen. a) Ein leerer Stack; b) die 3 wurde auf den Stack gepusht; c) Zustand des Stacks, nachdem die 4 gepusht wurde; d) der Stack nach Ausführung von +; e) der Stack nach Ausführung des .-Befehls.

Sie können jetzt verstehen, warum man den Stack auch als Stapelspeicher bezeichnet: Die Verhältnisse sind ähnlich wie bei einem Stapel von Essentabletts in einer Cafeteria oder Großküche. Wenn neue Tabletts auf den Stapel gelegt werden, dann wandern die alten nach unten, und die neuen liegen auf dem Stapel obenauf. Entnimmt jemand dem Stapel ein Tablett, so greift er auf das oberste und zieht es vom Stapel; ist das oberste Element weg, dann wandern die anderen Tabletts "um eine Position nach oben". Das zuvor zweite Element wird jetzt oberstes Element des Stapels.

Fahren wir jetzt in unserem Programm (1-3) fort. Wir treffen als nächstes auf das Symbol +. Hierbei handelt es sich nicht um ein Datenelement.

Das + ist vielmehr ein FORTH-Wort oder Befehl. FORTH kennt eine ganze Menge an Wörtern, die alle unterschiedliche Operationen auslösen. Wir wollen in diesem Text zur besseren Markierung der FORTH-Wörter **Fettdruck** verwenden. Das Wort **+** sorgt dafür, daß folgendes passiert: Zuerst werden die beiden obersten Elemente des Stacks entfernt. Für das Entfernen des obersten Elements hat sich eine eigene Terminologie eingebürgert; man spricht in diesem Zusammenhang von pop. Im FORTH-Jargon würde man also sagen, daß die beiden obersten Elemente vom Stack gepoppt werden. (Da immer nur das oberste Element des Stacks gepoppt werden kann, werden die Daten in der Reihenfolge "zuerst die Vier, dann die Drei" entfernt.) Als nächstes sorgt das Wort **+** dafür, daß die beiden Zahlen addiert werden. Dabei ergibt sich die Summe 7, die jetzt auf den Stack gepusht wird. Wir haben jetzt den Zustand in Abbildung 1-1d. Beachten Sie, daß die Vier und die Drei verschwunden sind und an ihrer Stelle jetzt der Wert 7 an oberster Stelle auf dem Stack steht.

Schließlich stoßen wir in unserem Programm auf das Wort **.** (.). Dieses Wort sorgt dafür, daß das oberste Element des Stacks gepoppt und auf Ihrem Terminal ausgegeben wird. Der Stack ist jetzt also leer, wie man an der Abbildung 1-1e erkennen kann. Nachdem FORTH die Zahl 7 auf Ihrem Terminal ausgegeben hat, bringt es die Meldung "ok" und zeigt somit an, daß das Programm ohne Fehler abgearbeitet werden konnte und der Computer jetzt auf neue Befehle von Ihnen wartet.

Wir kennen nun die beiden grundlegenden Stack-Operationen, nämlich das Abspeichern eines Datenelements als neues oberstes Element auf dem Stack ("push") und das Entfernen des obersten Stack-Elements ("pop"), wobei alle verbleibenden Elemente um eine Position weiterrücken. Die Stack-Operation bringt es mit sich, daß das zuerst auf dem Stack abgelegte Element das letzte ist, das von ihm entfernt werden kann. Man spricht in diesem Zusammenhang von dem Prinzip "Last In First Out" (übertragen etwa: als erstes rein, als letztes raus), das auch unter der Abkürzung LIFO bekannt ist. Ein anderer weit verbreiteter Name für den Stack lautet daher auch LIFO-Speicher. Wenn ein Datenelement einmal mittels Pop vom Stack entfernt worden ist, dann hat es der Computer "vergessen", d.h., es kann nicht mehr zu Berechnungen heran-

gezogen werden. In den folgenden Kapiteln werden wir deshalb noch Methoden kennenlernen, die es uns erlauben, ein Datenelement auf dem Stack mehrmals für Operationen heranzuziehen.

Sicher ist Ihnen aufgefallen, daß wir in FORTH zuerst die Daten eingeben, und dann das Symbol, das diese Daten manipuliert, wie in "3 4 + .". Die Reihenfolge ist am Anfang sicher etwas ungewöhnlich. Es hat sich aber gezeigt, daß sich dahinter ein äußerst wirkungsvolles Prinzip für die Programmierung von Computern verbirgt; auch begegnet uns diese Darstellungsweise bei einigen programmierbaren Taschenrechnern. Man bezeichnet diese FORTH-typische Schreibweise als Postfix-Notation oder umgekehrte polnische Notation. Letztere leitet sich von dem Erfinder dieser Schreibweise her, dem polnischen Logiker und Mathematiker J. Lukasiewicz. Die normale Schreibweise, die wir von der Schularithmetik her kennen, trägt den Namen Infix-Notation. Einer der wichtigsten Vorteile der Postfix-Notation besteht darin, daß man in ihr keine Klammern braucht. Aus der Schulalgebra wissen wir, daß in der Infix-Schreibweise häufig der Einsatz von Klammern nötig wird, um eindeutige Ausdrücke zu erzielen. Angenommen, wir wollen die Zahlen 3 und 5 addieren und dann das Ergebnis mit 10 multiplizieren. In der üblichen Infix-Schreibweise drücken wir das so aus:

(3 + 5) * 10 (1-5)

Beachten Sie, daß wir hier gleich das Symbol * verwendet haben, um die Multiplikation auszudrücken, ebenso, wie es in FORTH und den meisten anderen Programmiersprachen üblich ist. Wir brauchen die Klammern in (1-5), denn ohne sie würden wir die 3 auf das Produkt aus 5 und 10 addieren, wodurch wir ein völlig anderes Ergebnis erhalten.

Vergleichen wir jetzt einmal, wie man die Rechenaufgabe (1-5) in der Postfix-Notation hinschreibt. Wir machen uns das gleich an einem einfachen FORTH-Programm klar:

10 3 5 + * . (RETURN) (1-6)

Wie Sie sehen, sind hier keine Klammern nötig. Die Arbeitsweise dieses Programms wollen wir uns wieder an Hand einer graphischen Darstellung des Stacks (ein "Stackdiagramm") vergegenwärtigen (vgl. Abb. 1-2). In Abbildung 1-2a sehen wir den Stack, nachdem die Zahlen 10, 3 und 5 in dieser Reihenfolge gepusht wurden. Als nächstes taucht das Operationssymbol + auf. Deswegen werden die obersten beiden Zahlen 5 und 3 mittels Pop entfernt und das Ergebnis der Addition, die 8, statt dessen auf dem Stack abgelegt. Das Ergebnis sehen Sie in der Abbildung 1-2b. Als nächstes kommt das FORTH-Wort * an die Reihe. Dies poppt wiederum die zwei obersten Zahlen vom Stack und berechnet ihr Produkt. Das Ergebnis der Berechnung wird als neues oberstes Element auf den Stack gepusht, und wir erhalten Abbildung 1-2c. Wie Sie sehen können, haben die beiden Operationen jeweils zwei Zahlen vom Stack gepoppt und nur eine neue dafür abgelegt. Jedesmal, wenn eine Zahl mittels Push entfernt wurde, rücken die verbleibenden Daten auf dem Stack um eine Position nach oben (vgl. nochmal Abb. 1-2b). Schließlich kommt das FORTH-Wort . an die Reihe. Die oberste Zahl 80 wird gepusht und auf dem Bildschirm angezeigt. Als Ergebnis der Operation finden wir einen leeren Stack vor (Abb. 1-2d).

```
I----I    I----I    I----I    I----I
I  5 I    I  8 I    I 80 I    I    I
I----I    I----I    I----I    I----I
I  3 I    I 10 I    I    I    I    I
I----I    I----I    I----I    I----I
I 10 I    I    I    I    I    I    I
I----I    I----I    I----I    I----I
I    I    I    I    I    I    I    I
I----I    I----I    I----I    I----I
I    I    I    I    I    I    I    I
I----I    I----I    I----I    I----I
I .  I    I .  I    I .  I    I .  I
I .  I    I .  I    I .  I    I .  I
  .         .         .         .

  a)        b)        c)        d)
```

ABBILDUNG 1-2: Der Stack bei der Ausführung des Programms (1-6). a) Nachdem 10, 3 und 5 auf den Stack gepusht wurden; b) der Stack nach Ausführung von +; c) der Stack nach Ausführung von *; d) der Stack nach Ausführung von "." .

Bisher haben wir nur mit ganzen Zahlen (Fachausdruck:Integer) gerechnet. Integers sind solche Zahlen, die keine Nachkommastellen aufweisen und deshalb nicht als Bruchzahlen geschrieben werden müssen. Fürs erste werden wir uns in unseren Beispielprogrammen auch auf diesen Zahlentyp beschränken.

Dieser Abschnitt hat zwei wichtige Konzepte der Programmiersprache FORTH eingeführt: den Stack und die Postfix-Notation. Für das Programmieren in FORTH sind diese von zentraler Bedeutung. Außerdem haben wir die FORTH-Wörter für die Addition, die Multiplikation und die Ausgabe von Daten kennengelernt. Sie sollten jetzt bereits in der Lage sein, mit diesem Wissen ausgerüstet einige FORTH-Programme zu schreiben.

1.5 Wie läßt man ein FORTH-Programm laufen?

Einfache FORTH-Programme haben wir bereits kennengelernt. Diese werden von der Tastatur eingegeben und in dem Augenblick ausgeführt, in dem wir die Return-Taste drücken. Nach seiner Ausführung geht das Programm allerdings verloren. Wenn Sie es wieder laufen lassen wollen, dann müssen Sie es erneut in voller Länge eingeben. Deshalb wäre es bequem, wenn wir uns Programme so merken könnten, daß wir sie jederzeit ohne Neueingabe laufen lassen könnten. Wir behandeln in diesem Abschnitt die Möglichkeit zur Speicherung von Programmen auf Diskette oder Kassette. Da die dazu benötigten Arbeitsschritte nicht Teil der Sprache FORTH sind, sondern von Ihrem Betriebssystem abhängen, können wir auf dieses Thema nicht allzu detailliert eingehen. Genauere Einzelheiten dazu sollten Sie Ihrem FORTH-Handbuch entnehmen, das zusammen mit Ihrem System ausgeliefert wurde. Bei den meisten FORTH-Systemen ähneln sich jedoch die Techniken für das Eingeben, Bearbeiten und Speichern von Programmen. Wir wollen in diesem Abschnitt deshalb einen allgemeinen Überblick über die nötigen Arbeitsschritte geben. Außerdem untersuchen wir noch einmal genauer den Prozeß des "Laufenlassens" eines Programms.

Zur Eingabe längerer Programme bedient man sich meistens eines sog. Editors. Das ist ein spezielles Programm, mit dem man Textmaterial eingeben (in diesem Fall Ihr Programm) und bearbeiten (z.B. für Korrekturen) kann. Die meisten in FORTH-Systemen zur Verfügung stehenden Editoren sind sog. Bildschirmeditoren. Sie

bearbeiten stets eine sog. Bildschirmseite, also das, was Sie gerade auf dem Bildschirm Ihres Terminals zu sehen bekommen. In der Regel kann eine Bildschirmseite 1024 Zeichen aufnehmen, obwohl sich bei einigen größeren Bildschirmen ein anderer Textumfang ergibt. Meistens wird der Text auf 16 (oder 24) Zeilen dargestellt, von denen jede bis zu 64 (oder 80) Zeichen aufnehmen kann.

Die Programme werden auf Diskette oder Kassette gespeichert. Jeder Bildschirm wird innerhalb eines solchen Programms als ein <u>Block</u> bezeichnet. Die Blöcke sind fortlaufend durchnumeriert. Sie entsprechen bestimmten Adressen auf Ihrer Kassette oder Diskette. Die genauen Konventionen für die Vergabe von Blocknummern entnehmen Sie am besten Ihrem FORTH-Handbuch. Wenn Sie also beispielsweise ein Programm edieren und es anschließend in Block 120 abspeichern wollen, dann geben Sie in einem typischen FORTH-System folgendes ein:

```
120 EDIT (RETURN)                                          (1-7)
```

Als nächstes können Sie das Textmaterial eintippen, das Ihr Programm ausmacht. (Wenn Sie in einem Block arbeiten, der bereits einmal verwendet wurde, dann müssen Sie erst einmal das darin enthaltene Textmaterial löschen, ehe Sie mit der Neueingabe beginnen können.) In den folgenden Überlegungen gehen wir davon aus, daß das Programm auf einer Diskette gespeichert werden soll.

Der Block, der sich gerade in Arbeit befindet, wird nun nicht sofort auf Diskette gespeichert, sondern statt dessen in einen extra dafür reservierten Bereich im Arbeitsspeicher des Computers aufbewahrt. Diesen Spezialbereich bezeichnet man mit dem Fachausdruck <u>Puffer</u>. Nach Beendigung der Edierarbeit geben Sie ein spezielles Kommando ein, durch welches der Editor verlassen und der bearbeitete Puffer markiert wird, damit das System weiß, daß er <u>aktualisiert</u> werden muß. Der Computer weiß nun, daß der fragliche Puffer neues Textmaterial enthält, das noch nicht auf Diskette gespeichert worden ist; das Abspeichern neuen Materials auf die Diskette aber bezeichnet man als Aktualisieren. Jetzt könnten Sie einen anderen Block edieren. Das Programm könnte ja z.B. zu lang sein, um in einen einzigen Block zu passen. Dann setzen Sie es einfach in einem neuen, zweiten Block fort. Wenn Sie jetzt den Editor verlassen und den Block für die Aktualisierung markieren,

dann existieren somit im FORTH-System zwei Puffer, die jeweils neues Textmaterial enthalten und für die Aktualisierung markiert sind. Geben Sie jetzt ein

SAVE-BUFFERS(RETURN) (1-8a)

oder, bei einigen FORTH-Systemen

FLUSH(RETURN) (1-8b)

dann wird der Inhalt der markierten Puffer auf die Diskette gespeichert.

Da sich die Puffer im Hauptspeicher des Computers befinden, ist ihre Anzahl beschränkt. Wie viele Puffer genau in Ihrem FORTH-System zur Verfügung stehen, müssen Sie Ihrem Handbuch entnehmen. Bei vielen FORTH-Systemen können Sie dennoch mehr Blöcke bearbeiten, als Puffer zur Verfügung stehen. Wenn Sie einen neuen Block eröffnen wollen und kein Puffer mehr vorhanden ist, dann werden einfach alle markierten Puffer auf Diskette geschrieben und stehen danach wieder zur Verfügung. Dieser Prozeß der Aktualisierung geschieht völlig selbsttätig; der Benutzer braucht sich darum nicht zu kümmern. Durch das Speichern auf Diskette werden die Puffer wieder frei für die Aufnahme von neuem Textmaterial.

In vielen FORTH-Systemen sagen die Blocknummern auch aus, an welcher Stelle der Diskette sich der entsprechende Text befindet. Deshalb benötigen diese Systeme keine speziellen Inhaltsverzeichnisse für die Disketten. Da das Betriebssystem von FORTH in FORTH selbst geschrieben ist, ist es einfach, dieses zu ändern und gegebenenfalls Inhaltsverzeichnisse mit aufzunehmen, falls Sie dies wünschen. Wie dies geht, erfahren Sie in Kapitel 8.

Nehmen wir einmal an, Sie wollen ein Programm verändern, das Sie mit Ihrem Editor eingegeben haben. Das Programm soll sich im Block 120 befinden. Wieder geben Sie (1-7) ein. Daraufhin wird der Inhalt von Block 120 von der Diskette gelesen und auf Ihrem Bildschirm ausgegeben. Sie können jetzt eine Vielzahl unterschiedlicher Operationen mit diesem Text anstellen, wie z.B, Zeichen oder Zeilen einfügen oder löschen, bestehende Zeichen

überschreiben usw. Dadurch können Sie Ihr Programm verbessern, ohne daß Sie es völlig neu eingeben müssen. Zur Orientierung auf dem Bildschirm dient eine Schreibmarke, der sog. Cursor, der (meist in Form eines leuchtenden Rechtecks oder eines Unterstrichs) Ihnen anzeigt, an welcher Position in Ihrem Text Sie sich gerade befinden. Natürlich gibt es auch Kommandos, mit denen Sie diesen Cursor über Ihren Bildschirm bewegen und an jede beliebige Stelle des Texts gelangen können. Wenn Sie ein Zeichen tippen, erscheint dies immer an der Stelle, an der sich der Cursor gerade befindet. Die einzelnen Kommandos für die Textbearbeitung (oder "Editierung") sind natürlich von System zu System verschieden; ziehen Sie deshalb für die praktische Arbeit zuerst Ihr Handbuch zu Rate. Auch ist es vorteilhaft, wenn Sie erst ein wenig mit Ihrem Editor herumspielen, bevor Sie mit dem vorliegenden Buch weitermachen.

Angenommen, Sie haben im Editor ein Programm geschrieben; jetzt könnten Sie es eigentlich ausführen lassen. Wieder wollen wir davon ausgehen, daß sich das fragliche Programm im Block 120 befindet. Sie haben den Editor verlassen und den betreffenden Block für die Aktualisierung markiert. Als nächstes geben Sie ein

```
120 LOAD (RETURN)                                         (1-9)
```

Daraufhin wird das Programm ausgeführt. Beachten Sie, daß es sich nicht unbedingt in einem Puffer befinden muß. Wenn es bereits fertig geschrieben und abgespeichert ist, also auf der Diskette steht, dann sorgt der Befehl aus (1-9) dafür, daß das Programm in Block 120 automatisch von der Diskette gelesen und in einen Puffer geschrieben und dann aus diesem Puffer heraus ausgeführt wird. Voraussetzung dafür ist natürlich, daß Sie in Ihr Diskettenlaufwerk die richtige Diskette gelegt haben, die das gewünschte Programm enthält. Wenn FORTH einen Block lädt, dann compiliert es den Inhalt dieses Blockes erst in Maschinensprache und *führt dann das Ergebnis dieses Übersetzungsvorgangs aus.*

Oft beansprucht ein Programm mehr als einen Block. Dann müssen Sie, um das Programm laufen zu lassen, mehrere Blöcke laden. Das Programm könnte z.B. in den Blöcken 120 und 121 abgespeichert sein. Wenn Sie jetzt als letzte Zeile von Block 120 folgendes schreiben:

121 LOAD (1-10)

Dann weist dieser Befehl das FORTH-System an, den Block 121 zu laden. Sie brauchen also nur mehr Block 120 zu laden; der Befehl am Ende dieses Blockes sorgt dafür, daß Block 121 automatisch nachgeladen wird. Die genauen Einzelheiten dieses Verfahrens können auch wieder von System zu System variieren. Auch hierzu sollten Sie in Ihrem FORTH-Handbuch nachschlagen.

Sie haben jetzt einige grundlegende Techniken des Edierens und Ladens von Programmen kennengelernt. Kapitel 8 geht auf diese Arbeitsschritte noch einmal sehr ausführlich ein. Zusammen mit dem bisher Gesagten und Ihrem FORTH-Handbuch sollten Sie jedoch jetzt schon in der Lage sein, einfache FORTH-Programme zu schreiben, zu edieren und laufen zu lassen.

1.6 Programmfehler - Fehlersuche

So gut wie alle Programme enthalten anfänglich Fehler. Für den Anfänger ist dies oft entnervend, es sollte ihn jedoch nicht entmutigen. Erfahrene Programmierer wissen, daß die Fehlersuche ein fester Bestandteil des Programmmierprozesses ist; fehlerhafte Programme sind die Regel und nicht die Ausnahme. Sie weisen nicht auf ein Unvermögen des Programmierers hin. Es ist deshalb wichtig, daß man sich als Programmierer möglichst frühzeitig mit den Techniken zum Ausfindigmachen und Beheben von Fehlern in Programmen bekannt macht. Da Programmfehler im Programmiererjargon als Bugs bezeichnet werden (das kommt vom Englischen "bug" für "Wanze"), spricht man in diesem Zusammenhang auch von Debugging. Schon diese scherzhafte Ausdrucksweise, bei der die Fehlersuche unter "Entwanzen" läuft, sollte Ihnen klarmachen, daß Programmfehler keine Katastrophe sind.

Effiziente Fehlersuche setzt voraus, daß man sich über die Art möglicher Fehler im klaren ist. Die einfachste Art von Fehlern sind die sog. syntaktischen Fehler. Ein Syntaxfehler unterläuft Ihnen immer dann, wenn Sie sich beim Schreiben von FORTH-Programmen nicht an die formalen Regeln (die sog. "Syntax") halten, die für korrekte Programme in FORTH gelten. Wenn wir z.B. im Programm (1-6) das Leerzeichen zwischen dem + und dem * weglassen, dann führt dies zu einem Syntaxfehler. Wenn Sie versuchen, dieses

fehlerhafte Programm laufen zu lassen, dann erhalten Sie eine Fehlermeldung, in der Sie - soweit der Rechner dazu in der Lage ist - auf die Art Ihres Fehlers hingewiesen werden. Auf jeden Fall führt ein Syntaxfehler in den meisten FORTH-Systemen dazu, daß das Programm nicht ausgeführt und nicht die Bereitschaftsmeldung "ok" auf dem Bildschirm ausgegeben wird. Unterschiedliche FORTH-Systeme geben für den gleichen Fehler unterschiedliche Fehlermeldungen aus. Wenn Sie einen Block mit einem fehlerhaften Programm laden, so kann die Fehlermeldung z.B. die Programmzeile anzeigen, in der der Fehler aufgetreten ist. Meist bricht FORTH mit der Compilierung eines Programms ab, wenn es einmal einen Fehler in ihm entdeckt hat. Wie auch immer, bei fehlerhaften Programmen sollten Sie sofort das Programm edieren, den Fehler beseitigen und das Programm dann erneut laden. Jetzt funktioniert das Programm entweder richtig, oder Sie stoßen (an einer späteren Stelle) auf einen weiteren Syntaxfehler. Wenn Sie erst einmal die ungefähre Stelle des Syntaxfehlers wissen, dann können Sie ihn meist durch einfaches Durchlesen des Programms lokalisieren.

Ein anderer, etwas schwerer zu beseitigender Fehlertyp sind die <u>logischen Fehler</u>. Ein logischer Fehler liegt dann vor, wenn ein FORTH-Programm zwar allen formalen Regeln genügt, aber dennoch nicht tut, was Sie wollen. Wenn Sie z.B. im Programm 1-6 versehentlich an Stelle eines * ein + eingeben, dann berechnet Ihr Programm die Summe aus drei Zahlen, und nicht mehr, wie beabsichtigt, das Produkt aus einer Zahl und die Summe von zwei anderen. Das Programm ist zwar wohlgeformt, oder, wie man auch sagt, syntaktisch korrekt, liefert aber ein falsches Ergebnis. Jedesmal, wenn Sie ein neues Programm schreiben, sollten Sie deshalb - egal wie einfach das Programm ist! - anhand von Testdaten überprüfen, daß es auch das macht, was Sie wollen. Sie könnten z.B. die Ergebnisse Ihres Programms mit einem Taschenrechner überprüfen. Tun Sie das mit mehreren verschiedenen Eingabedaten. Bei der Überprüfung von Programmen kann man nie genug Sorgfalt walten lassen. <u>Überprüfen Sie also stets Ihre neuen Programme!</u>

Für die Lokalisation von Logikfehlern gibt es unterschiedliche Techniken. Als erstes sollten Sie Ihr Programm noch einmal durchlesen. Sollte der Logikfehler auf einen Schreibfehler zurückzuführen sein, dann fällt Ihnen dieser vielleicht beim Durchlesen schon auf. Gehen Sie Ihr Programm auch in Gedanken Schritt für Schritt durch, genauso, wie es der Computer tut. Dabei entdeckt man meist Fehler in der Ablauflogik: es zeigt sich, daß Ihr Programm gar nicht berechnet, was Sie berechnet haben wollen.

Zeichnen Sie auch Diagramme wie in den Abbildungen 1-1 oder 1-2, um sich den Zustand des Stacks in jedem Verarbeitungsschritt zu vergegenwärtigen. Auch dadurch werden einige Fehler offenbar. Einige FORTH-Systeme verfügen über Befehle, mit denen Sie den Stack ausdrucken können, ohne ihn zu verändern. Dies kann beim Debugging eine große Hilfe sein, da Sie dann nicht mehr von Hand jeweils die Stack-Werte notieren müssen, sondern einfach mit diesem Befehl nachsehen können, ob sich auch die gewünschten Werte an der richtigen Stelle im Stack befinden. Wie Sie ja bereits wissen, entfernt das FORTH-Wort . das oberste Element vom Stack und kann deshalb für diesen Zweck nicht eingesetzt werden. Im nächsten Kapitel werden wir übrigens eine Methode kennenlernen, die dieses Problem umgeht.

Einer der großen Vorteile von FORTH ist es, daß Sie in dieser Programmiersprache ein großes Programm schreiben können, das aus einer Folge von mehreren kleinen Unterprogrammen besteht. Je kleiner ein Programm ist, desto einfacher kann man in ihm logische Fehler lokalisieren und beseitigen. Jedes Unterprogramm wird sofort, nachdem es fertig geschrieben ist, erst einmal getestet und fehlerfrei gemacht. Dadurch bleibt der Prozeß der Fehlersuche stets auf relativ kleine und überschaubare Einheiten beschränkt. Wie man solche Unterprogramme schreibt, erfahren Sie im nächsten Kapitel.

1.7 Übungsaufgaben

1-1 Welche Funktion haben die einzelnen Bestandteile eines Rechners?

1-2 Was ist der Unterschied zwischen Maschinensprache und Assemblersprache?

1-3 Was ist der Unterschied zwischen einer Assemblersprache und einer höheren Programmiersprache?

1-4 Was ist ein Compiler?

1-5 Was ist ein Linker?

1-6 Was ist ein Interpreter?

1-7 Welche Vor- und Nachteile hat FORTH? Überwiegen die Vorteile die Nachteile?

1-8 Schreiben Sie ein FORTH-Programm, das fünf Zahlen addiert.

1-9 Schreiben Sie ein FORTH-Programm, das folgende Berechnung ausführt:

(3+4+5)6

1-10 Schreiben Sie ein FORTH-Programm, das folgende Berechnung ausführt

(5+6+7+20+21)(8)(9)

1-11 Schreiben Sie das Programm aus Übung 9 jetzt mit Ihrem Editor. Lassen Sie das Programm laufen.

1-12 Schreiben Sie das Programm aus Übung 10 jetzt mit Ihrem Editor. Lassen Sie das Programm laufen.

2
Grundlegende FORTH-Operationen

2 Grundlegende FORTH-Operationen

Dieses Kapitel erweitert unsere Kenntnis der FORTH-Operationen, basierend auf dem im letzten Kapitel besprochenen Stoff. Wir beginnen mit einer Erörterung der elementaren arithmetischen Operationen. Dabei beschäftigen wir uns auch noch einmal mit dem Stack und den Manipulationen, die mit dem Stack möglich sind. Aus Kapitel 1 kennen wir bereits das Konzept der FORTH-Wörter oder Kommandos; das vorliegende Kapitel geht auf diesen Begriff noch genauer ein und zeigt Ihnen auch, wie Sie eigene FORTH-Wörter definieren können. Dadurch können Sie ein größeres Programm in eine Folge kleinerer Unterprogramme aufteilen. Dies sollte Ihnen ermöglichen, bereits verhältnismäßig komplexe FORTH-Programme zu schreiben. Nocheinmal: Im laufenden Text (nicht in den Beispielprogrammen) schreiben wir FORTH-Wörter oder Kommandos in **Fettdruck**. Dadurch können Sie die zur Sprache FORTH gehörigen Ausdrücke leicht vom restlichen Text unterscheiden.

2.1 Arithmetische Operationen

Dieser Abschnitt behandelt die wichtigsten arithmetischen Operationen, nämlich die Addition, Subtraktion, Multiplikation und Division. Damit Sie in den für FORTH wichtigen Aspekten auch ganz sattelfest werden, wiederholen wir hier auch einige der Informationen, die bereits im letzten Kapitel gegeben wurden. Wir machen Sie auch mit einer praktischen Schreibweise vertraut, mit der man die Stack-Operationen ausdrücken kann.

2.1.1 Addition

Wenden wir uns zuerst der Addition zu. Das FORTH-Wort, das eine Addition bewirkt, ist das Pluszeichen (+). Wie Sie bereits aus dem letzten Kapitel wissen, sorgt + dafür, daß die zwei obersten Zahlen auf dem Stack von diesem entfernt werden (pop) und statt dessen ihre Summe auf den Stack gepusht wird. Wir wollen einmal sehen, wie wir diese Vorgänge im Stack symbolisch darstellen können. Angenommen, wir haben drei Zahlen n_3, n_2 und n_1, wobei n_3

das oberste Stack-Element ist, n_2 das nächste usw. Eine bildliche Darstellung dieser Situation sehen Sie in Abbildung 2-1. Zur Darstellung dieses Zustands schreiben wir ganz einfach

n_1 n_2 n_3

```
I-----I
I n3  I
I-----I
I n2  I
I-----I
I n1  I
I-----I
I     I
 I-----I
 I     I
 I-----I
 I  .  I
 I  .  I
    .
```

ABBILDUNG 2-1: Ein Stack mit drei Zahlen

Wie Sie sehen, wird der Inhalt des Stacks charakterisiert durch eine Folge von Zahlen, die durch Leerzeichen getrennt sind. Das am weitesten rechts stehende Element dieser Zahlenfolge stellt das oberste Stack-Element dar. Mit dieser Notation können wir den FORTH-Befehl + anschaulich charakterisieren.

n_1 n_2 --> n_{summe} (2-2)

Dabei soll n_{summe} die Summe der Zahlen n_1 und n_2 darstellen. Wir bezeichnen einen Ausdruck, wie ihn Beispiel 2-2 bringt, als Stack-Relation. Die Ausdrücke links vom Pfeil stellen den Zustand des Stacks vor Ausführung des FORTH-Befehls dar, während der Ausdruck rechts vom Pfeil den Zustand des Stacks nach der Ausführung des FORTH-Befehls repräsentiert. Natürlich ist es unnötig, den ganzen Stack aufzuschreiben; lediglich die an der Operation

beteiligten Werte sind von Interesse. Als Beispiel wollen wir ein kleines Programm schreiben, das fünf Zahlen addiert und ihre Summe ausgibt.

```
23 45 6 78 1 + + + + + . (RETURN)                    (2-3)
```

Auf dieses Programm reagiert der Computer mit

```
153 ok
```

In Abbildung 2-2 sehen Sie den Zustand des Stacks während verschiedener Schritte bei der Ausführung dieses Programms. Abbildung 2-2a zeigt den Stack, nachdem die Zahlen 23, 45, 6, 78 und 1 in dieser Reihenfolge gepusht worden sind. Nach Ausführung des ersten (linkesten) + werden die obersten zwei Stack-Einträge entfernt und durch ihre Summe ersetzt. Dies bedeutet, daß zwei Zahlen durch eine ersetzt werden, weswegen die verbleibenden Stack-Daten um eine Postition nach "oben" wandern. Diesen Zustand können Sie in Abbildung 2-2b sehen. Ähnlich zeigen die Abbildungen 2-2c, 2-2d und 2-2e den Stack nach Ausführung der zweiten, dritten und vierten Addition. Schließlich sorgt der Punktbefehl . dafür, daß das Ergebnis (das oberste Stack-Element) ausgedruckt und vom Stack entfernt wird. Nach Beendigung des Programms ist der Stack also wieder leer, so, wie Sie es in Abbildung 2-2f sehen können. Natürlich stimmt dies nur, wenn der Stack auch vor Ausführung des Programms bereits leer war.

2.1.2 Subtraktion

Das FORTH-Wort für die Subtraktion ist das Minuszeichen (-). Die Stack-Relation, die die Subtraktion charakterisiert, sieht folgendermaßen aus:

$$n_1\ n_2 \dashrightarrow n_{diff} \qquad (2\text{-}4)$$

```
I-----I   I-----I   I-----I   I-----I   I-----I   I-----I
I   1 I   I  79 I   I  85 I   I 130 I   I 153 I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I   I-----I
I  78 I   I   6 I   I  45 I   I  23 I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I   I-----I
I   6 I   I  45 I   I  23 I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I   I-----I
I  45 I   I  23 I   I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I   I-----I
I  23 I   I     I   I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I   I     I   I     I

   a)        b)        c)        d)        e)        f)
```

Abbildung 2-2: Zustand des Stacks während der Ausführung von Programm (2-3). a) Nachdem 23, 45, 6, 78 und 1 auf den Stack gepusht wurden; b) nach Ausführung des ersten +; c) nach Ausführung des zweiten +; d) nach Ausführung des dritten +; e) nach Ausführung des vierten +; f) nach Ausführung des Punktkommandos.

Dabei ergibt sich n_{diff} durch Subtraktion von n_1 minus n_2. Wie Sie sehen können, wird also das oberste Stack-Element vom zweiten Stack-Element subtrahiert. Ein Programm, das von der Zahl 5 die 3 subtrahiert und das Ergebnis ausdruckt, sieht folgendermaßen aus:

```
5 3 - . (RETURN)
```

In Abbildung 2-3 können Sie wieder sehen, wie sich der Stack bei Ausführung dieses Programms verhält.

Das Minuszeichen hat in FORTH eine doppelte Bedeutung; man kann es nämlich dazu benutzen, negative Zahlen einzugeben. Schreiben wir in FORTH das Minuszeichen vor eine Zahl (ohne Leerzeichen dazwischen!), so weiß FORTH, daß es sich dabei um eine negative

```
I-----I  I-----I  I-----I
I   3 I  I   2 I  I     I
I-----I  I-----I  I-----I
I   5 I  I     I  I     I
I-----I  I-----I  I-----I
I     I  I     I  I     I
I-----I  I-----I  I-----I
I  .  I  I  .  I  I  .  I
I  .  I  I  .  I  I  .  I
I  .  I  I  .  I  I  .  I
I     I  I     I  I     I

   a)       b)       c)
```

Abbildung 2-3: Zustand des Stacks bei Ausführung des Programms (2-5). a) Nachdem 5 und 3 auf den Stack gepusht wurden; b) nach Ausführung des -; c) nach Ausführung des Punktkommandos.

Zahl handelt. Das Minuszeichen ist in diesem Fall also kein FORTH-Befehl, vielmehr ist es "Teil" der Zahl. Deshalb führt das Programm

```
2 -5 - . (RETURN)
```

dazu, daß als Ergebnis 7 ausgegeben wird. Wie Sie sehen können, befindet sich zwischen dem Minuszeichen und der Ziffer 5 kein Leerzeichen. Bekanntermaßen sind alle Zahlen, mit denen wir es bisher zu tun gehabt haben, ganze Zahlen oder Integers. Integers sind, wie Sie bereits wissen, Zahlen ohne Nachkommastellen. Deshalb dürfen sie auch nicht mit einem Dezimalpunkt geschrieben werden. (Alle Computersprachen benutzen zur Darstellung von Nachkommastellen die amerikanische Schreibweise; FORTH bildet hier keine Ausnahme. In der amerikanischen Schreibweise wird das Dezimalkomma durch einen Dezimalpunkt dargestellt.) Die Integers, mit denen wir in FORTH arbeiten können, dürfen allerdings nicht beliebig groß sein. Sie müssen sich zwischen -32768 und 32767 einschließlich bewegen. Falls Ihr Programm eine Zahl berechnet, die sich nicht innerhalb dieses Bereichs befindet, dann erhalten Sie ein falsches Ergebnis!. Natürlich ist es auch möglich, Zahlen von

größerem Betrag bei Berechnungen heranzuziehen; wie dies geht, erfahren Sie in Kapitel 5.

2.1.3 Multiplikation

Multiplikation drückt man in FORTH mit dem Wort * aus. Bei Ausführung der Multiplikation werden die zwei obersten Stack-Elemente entfernt und miteinander multipliziert. Danach wird das Produkt auf den Stack gepusht. Die Stack-Relation für die Multiplikation lautet:

$n_1 \; n_2 \; \text{-->} \; n_{prod}$ (2-7)

Dabei ist n_{prod} das Produkt der Zahlen n_1 und n_2. Betrachten wir einmal das folgende FORTH-Programm:

```
24 5 3 - * . (RETURN)                                    (2-8)
```

Abbildung 2-4 zeigt Ihnen wieder den Stack bei den einzelnen Programmschritten. Abbildung 2-4a zeigt den Stack, nachdem 24, 5 und 3 in dieser Reihenfolge gepusht wurden. Nach Ausführung des - werden die obersten zwei Elemente gepoppt, wobei das oberste Stack-Element vom zweiten subtrahiert wird. Die Subtraktion hat das Ergebnis 2, welches nun als neues oberstes Element auf den Stack gepusht wird. Zuvor aber ist die 24 nach oben gewandert und ist somit das neue zweite Element auf dem Stack. Diesen Zustand zeigt die Abbildung 2-4b. Nun kommt die Multiplikation an die Reihe; wieder werden die beiden obersten Stack-Elemente entfernt und wir erhalten ihr Produkt (48). Diese Zahl wird auf den Stack gepusht, wodurch der Zustand in Abbildung 2-4d entsteht. Schließlich wird der Punktbefehl ausgeführt, das oberste Element gepoppt und ausgedruckt. Daraufhin ist der Stack leer (Abbildung 2-4d).

```
I-----I   I-----I   I-----I   I-----I
I   3 I   I   2 I   I  48 I   I     I
I-----I   I-----I   I-----I   I-----I
I   5 I   I  24 I   I     I   I     I
I-----I   I-----I   I-----I   I-----I
I  24 I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I
I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I
I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I

   a)        b)        c)        d)
```

ABBILDUNG 2-4: Der Stack für das Programm 2-8; a) nachdem 24, 5 und 3 in dieser Reihenfolge eingegeben wurden; b) nach Ausführung des -; c) nach Ausführung der Multiplikation; d) nach Ausführung des Punktkommandos.

2.1.4 Division

Bis jetzt haben wir uns in unseren Erörterungen auf Integers beschränkt, da dies die Darstellung vereinfachte. Wenn wir uns nun mit der Division befassen wollen, müssen wir der Arbeit mit Integers jedoch noch einmal unsere Aufmerksamkeit schenken. Betrachten Sie einmal die Division von 5/2; das Ergebnis ist 2.5. (Beachten Sie die amerikanische Darstellung von Bruchzahlen!) Wie Sie sehen, kann die Division zweier ganzer Zahlen ein nicht ganzzahliges Ergebnis haben. Was macht nun FORTH in solch einem Fall? Es läßt ganz einfach den Bruchteil des Ergebnisses weg. Wenn wir die obige Divisionsaufgabe unserem FORTH-System stellen, dann erhalten wir als Ergebnis die 2! Der Faktor 0,5 ist deshalb noch nicht verloren, denn es gibt in FORTH eine Möglichkeit, an den Divisionsrest bei ganzzahliger Division heranzukommen. In diesem Fall wird der Divisionsrest 1. (Es gibt natürlich auch die Möglichkeit, in FORTH mit Bruchzahlen zu arbeiten; dies besprechen wir jedoch erst in Kapitel 5.) Bei der Arbeit mit negativen Zah-

len verhält es sich genauso. So ergibt -14/3 den Quotienten -4 mit Rest -2/3. Das Divisionsergebnis ist also -4, die Nachkommastellen (2/3) werden einfach fallengelassen. Diese Division hat den Rest -2. Die Division von 14/(-3), die anscheinend gleich der obigen ist, führt dennoch auf ein anderes Erbgebnis. Wir können nämlich schreiben 14/(-3) = -4+(2/-3). In diesem Fall erhalten wir als Divisionsergebnis immer noch -4; jetzt ist der Divisionsrest jedoch 2. Probieren Sie ein wenig mit Ihrem FORTH-System herum, um herauszufinden, wie es sich bei Division mit negativen Zahlen verhält.

Wir kennen bisher noch nicht das FORTH-Wort für die Division; es ist der Schrägstrich. Die Stack-Relation für die Division lautet

$$n_1\ n_2 \dashrightarrow n_{quot} \qquad (2\text{-}9)$$

Auch die Division entfernt, wie alle anderen arithmetischen Operationen, die wir bisher kennengelernt haben, die obersten zwei Stack-Elemente. n_1 wird durch n_2 dividiert und der sich dabei ergebende Quotient auf den Stack gepusht. Beachten Sie, daß das zweite Stack-Element durch das erste Stack-Element dividiert wird. Bei Ausführung des FORTH-Wortes / wird kein Divisionsrest berechnet. Dazu kommen wir etwas später. Dividieren wir doch einmal 5 durch 3 und lassen uns das Ergebnis ausdrucken:

```
5 3 / . (RETURN)                                        (2-10)
```

Abbildung 2-5 zeigt das zu diesem Programm gehörende Stack-Diagramm. Das Programm 2-10 hat uns allerdings keinen Divisionsrest geliefert; das liegt daran, daß das FORTH-Wort / sich nicht um den Divisionsrest kümmert. Dafür gibt es ein spezielles Kommando in FORTH, nämlich das Wort **MOD**; dieses dient zur Berechnung des Rests einer Division. Die Stack-Relation für dieses Wort ist

$$n_1\ n_2 \dashrightarrow n_{rest} \qquad (2\text{-}11)$$

In diesem Fall ist n_{rest} der Rest aus der Division von n_1/n_2. Wie Sie wissen, befand sich vor Ausführung der Division die Zahl n_2

```
I-----I   I-----I   I-----I
I   3 I   I   1 I   I     I
I-----I   I-----I   I-----I
I   5 I   I     I   I     I
I-----I   I-----I   I-----I
I     I   I     I   I     I
I-----I   I-----I   I-----I
I     I   I     I   I     I
I     I   I     I   I     I
I     I   I     I   I     I
I     I   I     I   I     I
I     I   I     I   I     I
I     I   I     I   I     I

  a)        b)        c)
```

ABBILDUNG 2-5 Stack-Diagramm für Programm (2-10); a) nach Eingeben von 5 und 3 in dieser Reihenfolge; b) nach Ausführung von /; c) nach Ausführung des Punktkommandos.

an oberster Stelle des Stacks. Das Vorzeichen von n_{rest} ist, gemäß den Konventionen von FORTH-79, das gleiche wie das von n_1. Betrachten wir jetzt einmal das FORTH-Programm, das den Rest der Division von 5/3 liefert.

```
5 3 MOD . (RETURN)                                    (2-12)
```

In Abb. 2-6 finden Sie das zugehörige Stack-Diagramm.

Es wäre nun schön, wenn wir in einem Programm beide Größen, den Quotienten und den Rest, gleichzeitig berechnen könnten. Dafür gibt es glücklicherweise ein eigenes FORTH-Wort, nämlich **/MOD**. Beachten Sie, daß in diesem FORTH-Wort zwischen dem Schrägstrich und dem M kein Leerzeichen stehen darf. Die Stack-Relation, die dieses Wort charakterisiert, sieht folgendermaßen aus:

$$n_1 \; n_2 \; \text{-->} \; n_{rest} \; n_{quot} \qquad (2\text{-}13)$$

```
I-----I  I-----I  I-----I
I   3 I  I   2 I  I     I
I-----I  I-----I  I-----I
I   5 I  I     I  I     I
I-----I  I-----I  I-----I
I     I  I     I  I     I
I-----I  I-----I  I-----I
I     I  I     I  I     I
I     I  I     I  I     I
I     I  I     I  I     I
I     I  I     I  I     I
I     I  I     I  I     I
I     I  I     I  I     I
I     I  I     I  I     I
I     I  I     I  I     I

   a)       b)       c)
```

ABBILDUNG 2-6: Stack-Diagramm für Programm (2-12); a) nach Eingeben von 5 und 2 in dieser Reihenfolge; b) nach Ausführung des Wortes **MOD**; c) nach Ausführung des Punktkommandos.

Hier dividieren wir n_1 durch n_2 und entfernen diese beiden Zahlen vom Stack. Daraufhin werden der Divisionsrest n_{rest} und Quotient n_{quot} auf den Stack gepusht. Wie Sie sehen können, ist vor Ausführung der Operation n_2 das oberste Stack-Element, nach Ausführung von **/MOD** befindet sich n_{quot} an oberster Stelle des Stacks. Hierzu ein Beispiel:

5 3 /MOD . . (RETURN) (2-14)

Abbildung 2-7 zeigt das zugehörige Stack-Diagramm. Wie Sie sehen, wird nach Ausführung von **/MOD** der Quotient gedruckt und vom Stack entfernt. Deshalb wandert die 2 nach oben und wird neues oberstes Element. Nach Ausführung des zweiten Punktkommandos wird der Divisionsrest gedruckt und vom Stack entfernt.

```
I-----I   I-----I   I-----I   I-----I
I   3 I   I   1 I   I   2 I   I     I
I-----I   I-----I   I-----I   I-----I
I   5 I   I   2 I   I     I   I     I
I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I
I .   I   I     I   I     I   I     I
I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I

  a)        b)        c)        d)
```

ABBILDUNG 2-7: Stack-Diagramm für das Programm (2-14). a) Nach Eingeben von 5 und 3 in dieser Reihenfolge; b) nach Ausführung von **/MOD**; c) nach Ausführung des ersten Punktkommandos; d) nach Ausführung des zweiten Punktkommandos.

Wenn wir also das Programm 2-14 laufen lassen, dann erhalten wir folgenden Output

```
1 2 ok
```

Als letztes Beispiel wollen wir ein Programm schreiben, das den Quotienten und Divisionsrest von 3016/(12+3+14)5 berechnet. (Beachten Sie, daß die 5 im Nenner der Division steht.) Das zugehörige Programm sieht folgendermaßen aus:

```
3016 5 14 3 12 + + * /MOD . . (RETURN)                    (2-15)
```

Abbildung 2-8 ist das Stack-Diagramm zu diesem Programm. Abbildung 2-8a zeigt den Stack nach Eingabe von 3016, 5, 14, 3 und 12 in dieser Reihenfolge. In Abbildung 2-8b sehen wir den Stack,

nachdem die beiden obersten Elemente entfernt und durch ihre Summe (15) ersetzt worden sind. Wie Sie sehen können, sind die 14, 5 und 3016 jeweils um eine Position nach oben gewandert. Den Zustand des Stacks nach Ausführung des zweiten + zeigt die Abbildung 2-8c. In Abbildung 2-8d sehen Sie das Ergebnis der Multiplikation der beiden obersten Stack-Elemente 29 und 5; sie werden durch das Produkt 145 ersetzt, und die Zahl 3016 wandert nach oben. Abbildung 2-8f zeigt den Stack, nachdem die Zahlen 145 und 3016 gepoppt wurden und **/MOD** ausgeführt wurde. Jetzt befinden sich der Divisionsrest 116 und der Quotient 20 auf dem Stack, wobei der Quotient oberstes Stack-Element ist. Die Abbildung 2-8e und 2-8f zeigen den Stack zum Stand nach Ausführung der beiden Punktkommandos. Dieses Programm bringt folgendes Ergebnis auf den Bildschirm Ihres Terminals:

```
20 116 ok
```

Beachten Sie die Reihenfolge, in der wir die Daten für dieses Programm eingegeben haben. Wir wollten 12, 3 und 14 addieren. Deren Summe sollte als nächstes mit 5 multipliziert werden, und schließlich wollten wir 3016 durch das Ergebnis dieser Multiplikation dividieren. Da 3016 die letzte Zahl ist, mit der etwas geschehen soll, muß sie sich auch als unterstes Element auf dem Stack befinden. Deshalb geben wir diese Zahl als erste ein, denn jedesmal, wenn eine neue Zahl auf den Stack gepusht wird, wandert die 3016 um eine Position nach unten. Die 5 geht als vorletzte Zahl in die ganze Operation ein und wird deshalb als zweites Element gepusht. Dann geben wir 14, 3 und 12 ein. Wenden wir uns jetzt den FORTH-Wörtern in diesem Beispielprogramm zu. Das Programm soll als erstes die Summe der drei obersten Stack-Elemente berechnen. Deshalb stehen in dem Programm zuerst die beiden +-Wörter. Deren Summe wollen wir als nächstes mit 5 multiplizieren; deshalb geben wir das Kommando *. Das nächste FORTH-Wort ist **/MOD**. Dies sorgt dafür, daß 3016 durch das Berechnungsergebnis der letzten Operationen dividiert wird und sowohl Quotient als auch Divisionsrest auf den Stack gepusht werden. Schließlich brauchen wir noch zwei Punktkommandos, um Quotient und Rest auf den Bildschirm zu bringen.

```
I-----I  I-----I  I-----I  I-----I  I-----I  I-----I
I  12 I  I  15 I  I  29 I  I 145 I  I  20 I  I 116 I
I-----I  I-----I  I-----I  I-----I  I-----I  I-----I
I   3 I  I  14 I  I   5 I  I3016 I  I 116 I  I     I
I-----I  I-----I  I-----I  I-----I  I-----I  I-----I
I  14 I  I   5 I  I3016 I  I     I  I     I  I     I
I-----I  I-----I  I-----I  I-----I  I-----I  I-----I
I   5 I  I3016 I  I     I  I     I  I     I  I     I
I-----I  I-----I  I-----I  I-----I  I-----I  I-----I
I3016 I  I     I  I     I  I     I  I     I  I     I
I-----I  I-----I  I-----I  I-----I  I-----I  I-----I
I     I  I     I  I     I  I     I  I     I  I     I
I     I  I     I  I     I  I     I  I     I  I     I
I     I  I     I  I     I  I     I  I     I  I     I
I     I  I     I  I     I  I     I  I     I  I     I
I     I  I     I  I     I  I     I  I     I  I     I

   a)       b)       c)       d)       e)       f)

I-----I
I     I
I-----I
I     I
I-----I
I     I
I-----I
I     I
I-----I
I     I
I-----I
I     I
I     I
I     I

   g)
```

ABBILDUNG 2-8: Stack-Diagramm für Programm 2-15. a) Nach Eingabe von 3016, 5, 14, 3 und 12 in dieser Reihenfolge; b) nach Ausführung des ersten +; c) nach Ausführung des zweiten +; d) nach Ausführung von *; e) nach Ausführung von **/MOD**; f) nach Ausführung des ersten Punktkommandos; g) nach Ausführung des zweiten Punktkommandos.

2.2 Stackmanipulationen

Dieser Abschnitt widmet sich einigen FORTH-Wörtern, die es uns erlauben, den Stack zu manipulieren. Zuvor wollen wir uns jedoch überlegen, wozu diese Stackoperationen gebraucht werden könnten. Nehmen Sie einmal an, wir wollen das oberste Element des Stacks ausgeben, ohne es davon zu entfernen. Mit den bisherigen Sprachmitteln ist uns das nicht möglich. Weiterhin können wir mit den bisher vorgestellten FORTH-Kommandos auch nicht den Ausdruck (12+3+4)*5/6 berechnen, ohne Daten und Operationssymbole miteinander zu vermischen. Das ist zwar möglich, aber in diesem Fall können wir für diese Operation kein eigenes FORTH-Wort mehr definieren. (Mehr darüber im nächsten Abschnitt.) Es sieht zwar so aus, als wäre obiger Ausdruck in FORTH ganz leicht zu berechnen; bei genauerer Betrachtung stellen wir jedoch fest, daß wir die 6 nicht als oberstes Element auf den Stack bringen und deshalb die Division nicht wie verlangt ausgeführt werden kann. Darum müssen wir wissen, wie man den Stack manipulieren kann.

DUP - Dieses FORTH-Wort dupliziert das oberste Stack-Element. Es wird durch folgendes charakterisiert:

n --> n n (2-16)

Ein Anwendungsbeispiel für dieses Wort sowie das zugehörige Diagramm folgen:

5 6 9 DUP (RETURN) (2-17)

Angenommen, wir wollen das oberste Stack-Element drucken, ohne es dadurch für immer vom Stack zu entfernen. Das folgende kleine Programm tut genau dies:

DUP . (RETURN) (2-18)

```
I-----I   I-----I
I  9  I   I  9  I
I-----I   I-----I
I  6  I   I  9  I
I-----I   I-----I
I  5  I   I  6  I
I-----I   I-----I
I     I   I  5  I
I-----I   I-----I
I     I   I     I
I-----I   I-----I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I

   a)        b)
```

ABBILDUNG 2-9: Stack-Diagramm für das Programm 2-17; a) nach Eingeben von 5, 6 und 9 in dieser Reihenfolge; b) nach Ausführung von **DUP**.

Wie Sie sehen, sorgt **DUP** dafür, daß die oberste Zahl dupliziert wird. Wenn nun als nächstes das Punktkommando ausgeführt wird, dann wird diese Zahl ausgegeben und vom Stack gepoppt. Da sie aber zuvor dupliziert worden war, ist sie noch einmal auf dem Stack vorhanden. Somit hat dieses Programm dafür gesorgt, daß das oberste Stack-Element ausgedruckt wird, aber auf dem Stack verbleibt.

DROP - Das FORTH-Wort **DROP** sorgt dafür, daß das oberste Stack-Element gepoppt, aber nicht ausgegeben wird. Alle anderen Stack-Einträge wandern um eine Position nach oben. **DROP** wird charakterisiert durch die Stack-Relation

n --> (2-19)

Abbildung 2-10 zeigt den Stack bei den einzelnen Schritten des folgenden Programms:

2 5 1 DROP (RETURN) (2-20)

```
I-----I   I-----I
I  1  I   I  5  I
I-----I   I-----I
I  5  I   I  2  I
I-----I   I-----I
I  2  I   I     I
I-----I   I-----I
I     I   I     I
I-----I   I-----I
I     I   I     I
I  .  I   I  .  I
I  .  I   I  .  I
I  .  I   I  .  I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I

   a)        b)
```

ABBILDUNG 2-10: Stack-Diagramm für Programm 2-20. a) Nach Eingeben von 2,5 und 1 in dieser Reihenfolge; b) nach Ausführung von **DROP**.

Es mag so aussehen, als ob **DROP** ein völlig überflüssiges Wort wäre; wir werden aber noch sehen, daß es durchaus seinen Nutzen hat.

Es gibt Programme, die unaufhörlich neue Zahlen auf den Stack legen. Dies kann folgenschwere Konsequenzen haben. Der Stack ist nämlich nichts anderes als ein Bereich im Arbeitsspeicher Ihres Computers. Wenn Sie fortgesetzt neue Daten auf dem Stack abspeichern, dann wird er einmal voll werden. Der Versuch, in dieser Situation neue Datenelemente auf den Stack zu pushen hat zur Folge, daß dazu Speicherstellen verwendet werden, die gar nicht für den Stack vorgesehen sind!. Man spricht in diesem Fall von einem Stack-Überlauf. Es kann dann z.B. passieren, daß Sie die Speicherstellen überschreiben, in denen sich Ihr FORTH-System selbst befindet, und so das System "zum Absturz bringen". In diesem Fall

bleibt Ihnen oft nichts anderes übrig, als den Resetknopf zu drücken und Ihr System neu zu starten. Das bedeutet aber, daß alle bisher eingegebenen Daten und Programme unwiederbringlich verlorengehen. Mit Hilfe von **DROP** können Sie aber unnütze Einträge vom Stack entfernen, so seine Größe reduzieren und einen Stack-Überlauf vermeiden.

SWAP - Das FORTH-Wort **SWAP** vertauscht die obersten beiden Einträge auf dem Stack. **SWAP** wird durch folgende Stack-Relation charakterisiert.

$n_1\ n_2 \dashrightarrow n_2\ n_1$ (2-21)

Erinnern Sie sich noch an unser Problem, den Ausdruck (12+3+14)* 5/6 zu berechnen? Wir sind jetzt dazu in der Lage, nämlich über das folgende Programm:

```
6 5 14 3 12 + + * SWAP /MOD . . (RETURN)
```
(2-22)

In Abbildung 2-11 sehen Sie das Stack-Diagramm für dieses Programm. Die Teilabbildungen 2-11d und 2-11e zeigen die Wirkungsweise von **SWAP**. Wie Sie sehen, sorgt es dafür, daß die beiden obersten Stack-Elemente vertauscht werden. Beachten Sie, daß sich **SWAP** nur auf die obersten zwei Elemente auf dem Stack auswirkt. Enthält Ihr Stack mehr als zwei Einträge, so bleiben alle außer den obersten beiden von der Operation **SWAP** unberührt.

OVER - Ein weiteres nützliches Wort zur Stack-Manipulation ist **OVER.** Dieses Wort dupliziert das zweite Element auf dem Stack und bringt das Duplikat an die oberste Stack-Position. Wir können **OVER** durch folgende Stack-Relation beschreiben:

$n_1\ n_2 \dashrightarrow n_1\ n_2\ n_1$ (2-23)

Folgendes Programm ist ein Anwendungsbeispiel für **OVER.**

```
2 3 4 5 OVER (RETURN)
```
(2-24)

```
I-----I   I-----I   I-----I   I-----I   I-----I
I 12  I   I 15  I   I 29  I   I 145 I   I  6  I
I-----I   I-----I   I-----I   I-----I   I-----I
I  3  I   I 14  I   I  5  I   I  6  I   I 145 I
I-----I   I-----I   I-----I   I-----I   I-----I
I 14  I   I  5  I   I  6  I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I  5  I   I  6  I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I  6  I   I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I  .  I   I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I   I  .  I
I     I   I     I   I     I   I     I   I     I

  a)        b)        c)        d)        e)

I-----I   I-----I   I-----I
I 24  I   I   1 I   I     I
I-----I   I-----I   I-----I
I  1  I   I     I   I     I
I-----I   I-----I   I-----I
I     I   I     I   I     I
I-----I   I-----I   I-----I
I     I   I     I   I     I
I-----I   I-----I   I-----I
I     I   I     I   I     I
I-----I   I-----I   I-----I
I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I

  f)        g)        h)
```

ABBILDUNG 2-11: Stack-Diagramm für Programm (2-22). a) nach Eingabe von 6, 5, 14, 3 und 12 in dieser Reihenfolge; b) nach Ausführung des ersten +, c) nach Ausführung des zweiten +; d) nach Ausführung des *, e) nach Ausführung von **SWAP**; f) nach Ausführung von **/MOD**; g) nach Ausführung des ersten Punktkommandos; h) nach Ausführung des zweiten Punktkommandos.

Das Stack-Diagramm für dieses Programm sehen Sie in Abbildung 2-12. Weitere Anwendungsbeispiele für **OVER** folgen später.

```
I-----I   I-----I
I  5  I   I  4  I
I-----I   I-----I
I  4  I   I  5  I
I-----I   I-----I
I  3  I   I  4  I
I-----I   I-----I
I  2  I   I  3  I
I-----I   I-----I
I     I   I  2  I
I-----I   I-----I
I     I   I     I
I-----I   I-----I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I

   a)        b)
```

ABBILDUNG 2-12: Stack-Diagramm für Programm 2-24. a) Nach Eingabe von 2, 3, 4 und 5 in dieser Reihenfolge; b) nach Ausführung von **OVER**.

PICK - Bei dem FORTH-Wort **PICK** handelt es sich um eine generalisierte Form von **OVER**. Mittels **PICK** können Sie jede beliebige Zahl auf dem Stack als neues oberstes Stack-Element duplizieren. Bei der Ausführung von **PICK** wird die oberste Zahl auf dem Stack entfernt. Diese Zahl entscheidet dann, welcher Eintrag auf dem verbleibenden Stack dupliziert werden soll. Dies läßt sich durch folgende Stack-Relation ausdrücken:

n_1 --> nn1 (2-25)

Wir wollen uns die Funktionsweise von **PICK** anhand eines Beispiels verdeutlichen.

32 345 567 189 764 3 PICK (RETURN) (2-26)

Betrachten Sie dazu das Stack-Diagramm in Abbildung 2-13. Bei Ausführung von **PICK** wird die 3, das oberste Stack-Element, gepoppt. Als nächstes wird die Zahl, die drittes Stack-Element auf dem verbleibenden Stack ist, als neues oberstes Element dupliziert. **PICK** entscheidet also anhand des obersten Stack-Eintrags, welcher andere Eintrag dupliziert werden soll; dabei zählt dieses oberste Stack-Element nicht mit! Beachten Sie, daß **2 PICK** das gleiche bedeutet wie **OVER**.

```
I-----I   I-----I
I   3 I   I 567 I
I-----I   I-----I
I 764 I   I 764 I
I-----I   I-----I
I 189 I   I 189 I
I-----I   I-----I
I 567 I   I 567 I
I-----I   I-----I
I 345 I   I 345 I
I-----I   I-----I
I  32 I   I  32 I
I-----I   I-----I
I     I   I     I
I  a) I   I  b) I
```

ABBILDUNG 2-13: Stack-Diagramm für 2-26. a) Nach Eingabe von 32, 345, 567, 189, 764 und 3 in dieser Reihenfolge; b) nach Ausführung von **PICK**.

ROT - Das FORTH-Kommando **ROT** bringt das dritte Stack-Element an die oberste Stack-Position. Die Stack-Relation für **ROT** lautet:

$n_1\ n_2\ n_3 \text{ --> } n_2\ n_3\ n_1$ (2-27)

Hier ein Beispielprogramm für den Einsatz von **ROT**.

```
23 45 11 34 55 ROT (RETURN)                                   (2-28)
```

Die Abbildung 2-14 zeigt den Stack während der Ausführung dieses Programms. 2-14a stellt den Stack nach Eingabe der Daten dar. 2-14b zeigt, wie der Stack nach Ausführung des **ROT**-Wortes aussieht.

Wie Sie sehen können, wurde das dritte Stack-Element entfernt und auf den Stack gepusht. Es wurden keine Stack-Elemente dupliziert; auch bleibt die Anzahl der Stack-Einträge unverändert.

```
I-----I  I-----I
I  55 I  I  11 I
I-----I  I-----I
I  34 I  I  55 I
I-----I  I-----I
I  11 I  I  34 I
I-----I  I-----I
I  45 I  I  45 I
I-----I  I-----I
I  23 I  I  23 I
I-----I  I-----I
I     I  I     I
I-----I  I-----I
I  .  I  I  .  I
I  .  I  I  .  I
I  .  I  I  .  I
I     I  I     I

   a)       b)
```

ABBILDUNG 2-14: Stack-Diagramm für das Programm 2-28. a) Nach Eingabe von 23, 45, 11, 34 und 55 in dieser Reihenfolge; b) nach Ausführung von **ROT**.

ROLL - Bei **ROLL** handelt es sich um eine generalisierte Form von **ROT**. Mit **ROLL** können Sie ein beliebiges Stack-Element ohne Duplikation an die oberste Stack-Position bringen. **ROLL** benutzt - ähnlich wie **PICK** - das oberste Stack-Element, um herauszufinden,

welcher Stack-Eintrag an die oberste Stelle gebracht werden soll. **ROLL** läßt sich also durch folgende Stack-Relation beschreiben:

n_1 --> $n_n 2$ (2-29)

Betrachten wir dazu das folgende Programm:

```
41 456 23 17 56 34 4 ROLL (ENTER)                    (2-30)
```

Abbildung 2-15 zeigt das zugehörige Stack-Diagramm. Im Teildiagramm 2-15a sehen wir den Stack nach Eingabe von 41, 456, 23, 17, 56, 34 und 4 in dieser Reihenfolge. Abbildung 2-15b zeigt den Stack nach Ausführung von **ROLL**. Das oberste Stack-Element, in diesem Fall die 4, wird vom Stack gepoppt und steuert die nachfolgende Operation von **ROLL**. Es wird nämlich der vierte Eintrag auf dem verbleibenden Stack, die 23, an oberste Position gebracht, und die Elemente unterhalb des vierten füllen durch "Aufrücken" die entstandene Lücke wieder auf. Wie Sie sehen können, ist **3 ROLL** äquivalent zu **ROT**.

DEPTH - Manchmal ist es wichtig zu wissen, wie viele Elemente sich auf dem Stack befinden. Das FORTH-Kommando **DEPTH** pusht eine Zahl auf den Stack, die diese Information liefert. Bei der Ermittlung der Stack-Tiefe (das englische Wort "depth" bedeutet Tiefe) zählt das Ergebnis dieser Operation nicht selbst zum Stack. Die Stack-Relation sieht folgendermaßen aus:

--> n_{tiefe} (2-31)

Beachten Sie, daß n_{tiefe} die Stack-Tiefe <u>vor</u> Ausführung des FORTH-Wortes **DEPTH** angibt, also den Zustand des Stacks, bevor n_{tiefe} selbst auf den Stack gebracht wurde. Sehen wir uns einmal ein Beispiel für den Einsatz von **DEPTH** an.

```
23 46 57 (RETURN)
52 437 56 78 (RETURN)                                (2-32)
DEPTH . (RETURN)
```

```
I-----I   I-----I
I   4 I   I  23 I
I-----I   I-----I
I  34 I   I  34 I
I-----I   I-----I
I  56 I   I  56 I
I-----I   I-----I
I  17 I   I  17 I
I-----I   I-----I
I  23 I   I 456 I
I-----I   I-----I
I 456 I   I  41 I
I-----I   I-----I
I  41 I   I     I
I-----I   I-----I
I     I   I     I
I-----I   I-----I
I     I   I     I
I     I   I     I
I     I   I     I
I     I   I     I

  a)        b)
```

ABBILDUNG 2-15: Stack-Diagramm für Programm (2-30). a) Nach Eingabe von 41, 456, 23, 17, 56, 34 und 4 in dieser Reihenfolge; b) nach Ausführung von **ROT**.

Abbildung 2-16 ist das zugehörige Stack-Diagramm. Erst einmal geben wir 23, 46 und 57 in dieser Reihenfolge ein. Daraus ergibt sich ein Stack-Zustand wie in Abbildung 2-16a. Als nächstes geben wir 52, 437, 56 und 78 ein. Die bisher auf dem Stack befindlichen Zahlen werden durch Eingeben dieser vier neuen Werte um insgesamt vier Positionen nach unten "gedrückt". Als nächstes führen wir **DEPTH** aus. Da sich auf dem Stack jetzt insgesamt 7 Zahlen befinden, wird auch eine 7 auf den Stack gepusht; dies sehen Sie in Abbildung 2-16c. Schließlich sorgt das Punktkommando dafür, daß die 7 vom Stack gepoppt und auf Ihrem Terminal ausgegeben wird. Danach ergibt sich der Stack aus der Abbildung 2-16d. Die Teilabbildungen 2-16b und 2-16d sind ganz offensichtlich identisch. Die Ausführung von **DEPTH** und anschließendes Ausdrucken dieses Ergebnisses ändern also nichts am Stack-Zustand.

```
I-----I   I-----I   I-----I   I-----I
I 57  I   I  78 I   I   7 I   I  78 I
I-----I   I-----I   I-----I   I-----I
I 46  I   I  56 I   I  78 I   I  56 I
I-----I   I-----I   I-----I   I-----I
I 23  I   I 437 I   I  56 I   I 437 I
I-----I   I-----I   I-----I   I-----I
I     I   I  52 I   I 437 I   I  52 I
I-----I   I-----I   I-----I   I-----I
I     I   I  57 I   I  52 I   I  57 I
I-----I   I-----I   I-----I   I-----I
I     I   I  46 I   I  57 I   I  46 I
I-----I   I-----I   I-----I   I-----I
I     I   I  23 I   I  46 I   I  23 I
I-----I   I-----I   I-----I   I-----I
I     I   I     I   I  23 I   I     I
I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I
I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I

  a)        b)        c)        d)
```

ABBILDUNG 2-16: Stack-Diagramm für Programm 2-30. a) nach Eingabe von 23, 46 und 57 in dieser Reihenfolge; b) nach Eingabe von 52, 437, 56 und 78 in dieser Reihenfolge; c) nach Ausführung von **DEPTH**; d) nach Ausführung des Punktkommandos.

2.3 Definieren eigener FORTH-Wörter

Die bisher besprochenen FORTH-Wörter werden beim Kauf Ihres Systems mit ausgeliefert, sind also schon herstellerseitig eingebaut. Einer der größten Vorteile von FORTH besteht aber darin, daß Sie Ihre eigenen Wörter schreiben und diese in den FORTH-Wörterschatz mit aufnehmen können. Dieser Abschnitt widmet sich diesen benutzerdefinierten FORTH-Wörtern. Sie versetzen uns in die Lage, umfangreiche und komplizierte Programme zu schreiben.

Zum Definieren eines eigenen FORTH-Wortes verwenden Sie die Kommandos : sowie ;. Wir wollen uns dies an einem Beispiel deutlich machen. Dazu definieren wir ein neues FORTH-Wort mit dem Namen **ADD3**, welches drei Zahlen addiert und die Summe auf dem Bildschirm ausgibt. Die Definition geht ganz einfach:

```
: ADD3 + + . ; (RETURN)                                    (2-33)
```

Sehen wir uns einmal genauer an, was wir eben gemacht haben. Wir haben mit dem Kommando : begonnen, auf das ein Leerzeichen folgt. An das Doppelpunktkommando schließt sich der Name des neu definierten FORTH-Wortes an, in diesem Falle **ADD3**. Es folgen die gewünschten Operationen, die durch ein oder mehrere Leerzeichen getrennt sind. Die Definition wird durch ein Leerzeichen und ein nachfolgendes ; abgeschlossen. Nachdem wir die Informationen in (2-33) eingegeben haben, gibt es ein neues FORTH-Wort: unser **ADD3**! Wenn wir jetzt irgendwann eintippen:

ADD3 (RETURN) (2-34)

dann ist dies genauso, als hätten wir all die Kommandos getippt, die in der Definition hinter dem Namen des neuen FORTH-Wortes (bis zum Strichpunkt) stehen. Es ist sehr wichtig, daß Sie hinter dem Doppelpunkt, der die Definition einleitet, ein Leerzeichen freilassen, und ebenso vor dem Strichpunkt, der die Definition abschließt. Die Art und Weise, ein neues FORTH-Wort zu definieren, welche wir im Beispiel (2-33) kennengelernt haben, trägt den Namen Doppelpunkt-Definition. Sie bewirkt, daß wir **ADD3** genauso wie jedes andere FORTH-Wort benutzen können. Allerdings geht das benutzerdefinierte Wort **ADD3** bei einem Warmstart des Computers verloren. Wir könnten dann zwar (2-33) erneut eingeben und so unser neues Wort wieder ins "Gedächtnis" von FORTH aufnehmen; bequemer aber ist es, mit dem Editor die Definition (2-33) in einem Block abzuspeichern. Jedesmal, wenn dieser Block geladen wird, wird dann automatisch **ADD3** zu einem Teil des FORTH-Systems.

Wenn wir jetzt eingeben:

```
27 34 56 ADD3 (RETURN)                                     (2-35)
```

dann ist das genau dasselbe, als hätten wir geschrieben

27 34 56 + + . (RETURN) (2-36)

In beiden Fällen erhalten wir als Ergebnis

117 ok

Noch einmal: Die Definition eines eigenen FORTH-Wortes muß eingeleitet werden durch das Wort :, gefolgt von einem Leerzeichen und abgeschlossen werden durch das Wort ;, dem ein Leerzeichen vorausgeht.

Bei der Definition eines neuen Wortes können Sie auf alle FORTH-Wörter zurückgreifen, die fest eingebauter Bestandteil des Systems sind, zusätzlich aber auch auf alle Wörter, die Sie selbst bereits definiert haben. Es gilt also die Regel: Jedes Wort, das in einer neuen Definition benutzt wird, muß zu diesem Zeitpunkt dem System bereits bekannt sein. Die eingebauten Wörter aus dem "Grundwörterschatz" sind dies sowieso; außerdem aber ist dem System jedes in der Zwischenzeit hinzugekommene benutzerdefinierte Wort bekannt und kann in Definitionen eingehen. Die Definition eines neuen Wortes - ob durch Eingabe vom Terminal oder durch Laden des entsprechenden Blockes - führt dazu, daß dieses Wort compiliert wird. Wenn Sie also z.B. die Definition (2-33) eingeben, so wird in dem Augenblick, in dem Sie die Return-Taste drükken, das neue Wort **ADD3** compiliert und dem System hinzugefügt.

Kommen in der Definition eines neuen Wortes andere benutzerdefinierte Wörter vor, dann müssen Sie - gemäß dem soeben Gesagten - diese zuvor Ihrem System bekannt machen. Wenn z.B. eines der Wörter in der neuen Definition in einem Block definiert ist, dann müssen Sie erst den Block laden, bevor Sie versuchen können, dieses Wort zu benutzen. Die Informationen in einem Block werden stets in der Reihenfolge geladen, in der sie sich im Block befinden. Sie können also in einem Block zwei Wörter definieren, wobei die zweite Definition bereits auf das erste Wort zurückgreift. Wie bereits erwähnt, wird ein fester Grundstock an "eingebauten" Wörtern bei jedem Laden des Systems mitgeladen. Einige Systeme verfügen aber noch über zusätzliche Blöcke mit sogenannten Erweiterungsworten. Diese Sonderbefehle werden nicht automatisch beim Laden des Systems bereitgestellt. Benötigen Sie diese Wörter, sei

es für eine Berechnung oder zur Definition eines neuen eigenen Wortes, dann müssen Sie erst den oder die Blöcke laden, in denen sich die speziellen Wörter befinden. Sie verhalten sich also genauso wie selbstdefinierte Wörter.

FORTH-Programme bestehen in der Regel aus einer Vielzahl selbstdefinierter Wörter. Ein Wort benutzt dabei meistens ein oder mehrere andere zuvor definierte Wörter, die ihrerseits wieder auf andere Wörter zurückgreifen können. Man sagt in diesem Zusammenhang auch, daß ein Wort ein anderes aufruft. Dieses gegenseitige Aufrufen von Wörtern ist eines der wichtigsten Charakteristika von FORTH. Sie sollten die einzelnen Definitionen möglichst kurz halten, um die Fehlersuche und Fehlerbeseitigung dadurch zu vereinfachen.

Zur Illustration des eben Gesagten wollen wir jetzt ein kurzes Programm schreiben, das den folgenden Ausdruck für beliebige Werte von x berechnet:

$$3x^4 + 2x^3 + 5x^2 + 2x + 4 \tag{2-37}$$

Abbildung 2-17 zeigt das zugehörige FORTH-Programm. Die Zahlen von 0 bis 15 am linken Rand sind nicht Teil des FORTH-Programms; es handelt sich dabei um sogenannte Zeilennummern, die Sie nur zu sehen bekommen, wenn Sie sich den Block auflisten lassen, der Ihr Programm enthält. Beim Schreiben oder Edieren eines Programms bekommen Sie diese Zeilennummern noch nicht zu sehen. Wir nehmen sie allerdings in unsere Programmlistings mit auf, um uns besser auf einzelne Teile des Programms beziehen zu können. Wenn sich unser Programm beispielsweise im Block 115 befindet, dann sorgen wir mit

```
115 LIST                                                    (2-38)
```

dafür, daß wir das Listing der Abbildung 2-17 erhalten.

```
 0 ( ZWEITE, DRITTE, VIERTE und POLYNOM )
 1 : ZWEITE DUP * ;
 2 : DRITTE DUP   ZWEITE    *  ;
 3 : VIERTE  DUP  DRITTE     *  ;
 4 :
 5 : POLYNOM      DUP DUP DUP
 6 :              VIERTE 3 *
 7 :              SWAP DRITTE 2 * +
 8 :              SWAP ZWEITE 5 * +
 9 :              SWAP 2 * +
10 :              4 + . ;
11
12
13
14
15
```

ABBILDUNG 2-17: Ein FORTH-Programm zur Berechnung des Polynoms (2-37) für verschiedene Werte von x; das Programm enthält auch die Definitionen von **ZWEITE, DRITTE** und **VIERTE.**

Gehen wir nun einmal die Einzelheiten des Programms der Abbildung 2-17 durch. Die Zeile 0 ist ein sogenannter Kommentar. Kommentare werden vom FORTH-System ignoriert. Ihr einziger Zweck besteht darin, den Programmierer oder andere Leser des Programms mit Informationen zu versorgen. Kommentare sind besonders dann für Sie sehr hilfreich, wenn Sie sich längere Zeit nach Definition eines Programms wieder mit diesem auseinandersetzen müssen. Kommentare können an beliebiger Stelle in einem FORTH-Wort oder -Programm auftauchen. Zum Schreiben eines Kommentars tippen Sie lediglich eine öffnende Klammer gefolgt von einem Leerzeichen. Der Text bis zur nächsten schließenden Klammer wird von FORTH ignoriert, wenn Sie das Programm compilieren lassen.

Betrachten wir nun Zeile 1. Hier definieren wir ein neues FORTH-Wort mit dem Namen **ZWEITE,** welches das oberste Stack-Element quadriert, oder, wie man auch sagt, seine "zweite Potenz" liefert. (Die zweite Potenz bzw. das Quadrat einer Zahl x schreibt man als x^2, was nichts anderes als x mit sich selbst multipliziert oder x mal x ist). **ZWEITE** dupliziert als erstes die Zahl, die sich zuoberst auf dem Stack befindet. Dann werden die beiden obersten Stack-Einträge (also zweimal dieselbe Zahl) vom Stack gepoppt und miteinander multipliziert. Wir erhalten somit die gewünschte Quadratzahl, die jetzt auf den Stack gepusht wird.

Nun zu Zeile 2 in Abbildung 2-17. Hier sehen wir die Definition des Wortes **DRITTE**, welches dafür sorgt, daß das oberste Stack-Element entfernt und durch seine dritte Potenz ersetzt wird. (Die dritte Potenz einer Zahl ist diese dreimal mit sich selbst multipliziert, also x mal x mal x). Im ersten Schritt wird das oberste Stack-Element dupliziert. Dann rufen wir unser selbstdefiniertes **ZWEITE** auf. Infolgedessen haben wir als oberstes Stack-Element das Quadrat der Ausgangszahl. Beachten sie, daß **ZWEITE** den restlichen Stack unverändert läßt. Deshalb ist das zweite Stack-Element immer noch die Ausgangszahl. Wenn wir nun den * ausführen, dann werden diese beiden Zahlen vom Stack gepoppt und miteinander multipliziert. Als Ergebnis erhalten wir die vierte Potenz der Ausgangszahl, die jetzt auf den Stack gepusht wird.

Zeile 3 der Abbildung 2-17 definiert in bereits bekannter Manier das Wort **VIERTE**, welches das oberste Stack-Element entfernt und durch seine vierte Potenz ersetzt. (Die vierte Potenz einer Zahl x ist x viermal mit sich selbst multipliziert, also x mal x mal x mal x). Die Definition von **VIERTE** lehnt sich sehr stark an die von **DRITTE** an, außer, daß **VIERTE** den gleichen Gebrauch von **DRITTE** macht, wie wir ihn von **DRITTE** und **ZWEITE** bereits kennen.

Wie Sie sehen können, haben wir Zeile 4 leer gelassen. Dem FORTH-Compiler machen Leerzeilen nichts aus. Leerzeilen dienen ebenso wie ein oder mehrere Leerzeichen in FORTH als Trenner zwischen den einzelnen Befehlen. Zusätzliche Leerzeichen und Leerzeilen erfüllen nur den Zweck, eine Definition für den menschlichen Leser klarer und deutlicher zu gestalten. Aus der Sicht von FORTH hätten wir genausogut die Definitionen für **ZWEITE**, **DRITTE** und **VIERTE** auf eine einzige Zeile quetschen können, so lange wir nur zwischen den einzelnen Wörtern immer mindestens ein Leerzeichen frei lassen. Die Lesbarkeit eines Programms kümmert den FORTH-Compiler nicht; er behandelt eine Folge von Zeilen in einem oder mehreren Blöcken als zusammenhängenden Text. Deshalb sind zusätzliche Leerzeichen und Leerzeilen nicht unbedingt erforderlich. Sie sollten sie jedoch freizügig einstreuen, um die Lesbarkeit Ihres Programms zu steigern. Gut lesbare Programme lassen sich viel leichter fehlerfrei machen. Ein sauberer Programmierstil hilft außerdem, Fehler zu vermeiden.

Zeile 5 der Abbildung 2-17 definiert nun das Kommando **POLYNOM**, mit dem man Ausdrücke der Form (2-37) berechnen kann. Setzen wir für x den Wert 5 ein, so geben wir ein:

5 POLYNOM (2-39)

Nun zu den Details von **POLYNOM.** Zuerst erzeugen wir drei Duplikate der obersten Stack-Zahl. Somit nimmt die fragliche Zahl die ersten vier Positionen auf dem Stack ein. Nun rufen wir **VIERTE,** woraufhin das oberste Stack-Element entfernt und durch seine vierte Potenz ersetzt wird. Daraufhin wird eine 3 auf den Stack gepusht. Die obersten zwei Zahlen werden nun gepoppt, miteinander multipliziert und ihr Produkt auf den Stack gepusht. Demnach ist das neue oberste Stack-Element nichts anderes als $3x^4$. In Zeile 7 rufen wir als nächstes das Kommando **SWAP.** Es sorgt dafür, daß die Ausgangszahl (5) zuoberst auf den Stack gebracht wird. Dann folgt das Kommando **DRITTE,** das gemäß seiner Definition x^3 auf dem Stack ablegt. Wir pushen nun die 2 auf den Stack und rufen das Wort *****. Jetzt befindet sich $2x^3$ zuoberst auf dem Stack, gefolgt von $3x^4$. Wenn nun durch + addiert wird, dann verschwinden diese beiden Zahlen vom Stack und werden durch ihre Summe ($3x^4 + 2x^3$) ersetzt. Machen Sie sich klar, daß jetzt die zweite Zahl auf dem Stack wieder die Ausgangszahl ist. Zeile 8 des Programms enthält wieder einen **SWAP**-Befehl, der erneut die Ausgangszahl nach oben bringt. Nach Ausführung von **ZWEITE** wird diese Zahl auf dem Stack durch ihr Quadrat ersetzt. Wir legen jetzt die 5 auf den Stack und duplizieren erneut. Daraufhin befindet sich $5x^2$ an oberster Stack-Position. Führen wir nun + aus, dann werden die beiden obersten Zahlen vom Stack entfernt und durch ihre Summe ersetzt, welche jetzt den Wert des Ausdrucks $3x^4 + 2x^3 + 5x^2$ darstellt. Wieder bringt ein **SWAP** aus Zeile 9 die Originalzahl zuoberst auf den Stack. Sie wird dann mit 2 multipliziert und das Ergebnis auf die zweite Zahl im Stack addiert. Darauf addieren wir dann noch in Zeile 10 die Zahl 4 und erhalten so das Endergebnis, welches wir noch ausgeben lassen.

Nach Laden des Blockes in Abbildung 2-17 können wir nicht nur das neue Wort **POLYNOM** überall verwenden, ebenso sind die Befehle **ZWEITE, DRITTE** und **VIERTE** jetzt verfügbar. Wenn wir z.B. eingeben

```
5 DRITTE . (RETURN)
```

dann sehen wir auf unserem Terminal

```
125 ok
```

Wenn also einmal der Block aus der Abbildung 2-17 geladen ist, dann können alle darin enthaltenen neu definierten FORTH-Wörter genauso benutzt werden, als wären sie eingebaute Wörter.

2.4 Mehr über FORTH-Wörter

Im letzten Absatz haben wir die Grundtechniken eingeführt, über die man zum Schreiben eigener FORTH-Wörter verfügen muß. Dieser Abschnitt beschäftigt sich nun mit einigen weiteren Details, die die Definition von FORTH-Wörtern betreffen. Jedes verwendbare FORTH-Wort (sei es nun in das System eingebaut oder nachträglich definiert bzw. geladen worden) ist im Hauptspeicher Ihres Computers abgelegt. Die Befehle, die das einzelne Wort ausmachen, werden dabei hintereinander in aufeinanderfolgenden Speicheradressen gespeichert. Der Speicherplatzbedarf eines FORTH-Wortes hängt natürlich von seiner Komplexität ab. Jedesmal, wenn Sie ein FORTH-Wort aufrufen, wird der Kontrollfluß Ihres Systems beeinflußt von den Instruktionen, die für das betreffende Wort im Computerspeicher enthalten sind. Der Speicherbereich des FORTH-Systems, in dem sich diese Instruktionen befinden, heißt Wörterbuch. Jeder Eintrag im Wörterbuch setzt sich aus mehreren Komponenten zusammen. Zusätzlich zu den Befehlen, die die Definition des einzelnen Wortes ausmachen, enthält ein Eintrag auch noch eine Repräsentation des Namens, den der Benutzer diesem Wort gegeben hat. Der Name dient zur Identifikation des Wortes. Außerdem enthält der Wörterbucheintrag die Anfangsadresse des Wortes, das unmittelbar vor dem aktuellen Wort compiliert wurde. FORTH merkt sich also immer nur die Adresse des Wörterbucheintrags, der vor der letzten Neudefinition compiliert wurde. Es hat den Anschein, als könnte das System mit dieser Methode immer nur den letzten Eintrag im Wörterbuch finden. Wir haben aber gesagt, daß jeder Wörterbucheintrag zusätzlich noch die Adresse des zuvor definierten Wortes enthält. Wenn Sie ein FORTH-Wort aufrufen, dann überprüft das System den Namenteil des letzten Wörterbucheintrags, um zu sehen, ob dieser Eintrag das gewünschte Wort enthält. Falls ja, dann wird die Programmsteuerung an die Instruktionen übergeben, die die Definition des Wortes ausmachen. Ist das gewünschte Wort aber nicht das letzte im Wörterbuch, dann wird der vorletzte Wörterbucheintrag untersucht. Das System kann diesen Eintrag finden, weil es sich ja beim letzten Wörterbucheintrag einen Verweis auf den vorletzten Eintrag gemerkt hat.

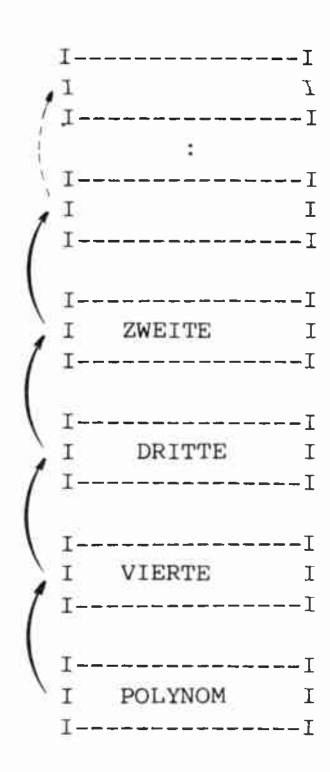


ABBILDUNG 2-18: Graphische Darstellung des Wörterbuches

Jetzt beginnt die Prozedur von neuem: Das neue Wort (welches im gesamten Wörterbuch das vorletzte ist) wird wieder untersucht, ob es das gewünschte ist. Falls ja, erhält es die Kontrolle, wird also ausgeführt, falls nein, geht FORTH dem bei diesem Wort gespeicherten Rückwärtsverweis nach und gelangt so zum drittletzten Eintrag im Wörterbuch. Wir sehen also, daß FORTH das Wörterbuch so lange von hinten nach vorne durchsucht, bis der fragliche Eintrag gefunden wird. Man spricht in diesem Zusammenhang von der Wörterbuchsuche. Befindet sich das Wort allerdings nicht im Wörterbuch, dann erhält der Benutzer eine Fehlermeldung, und der Prozeß des Compilierens bricht ab. Es hat den Anschein, als ob diese Form der Wörterbuchsuche äußerst umständlich sei; in Wirklichkeit geht sie aber sehr schnell vor sich.

Abbildung 2-18 ist eine graphische Darstellung des Zustands unseres Wörterbuches, nachdem der Block aus Abbildung 2-17 in das System geladen wurde. Wie Sie sehen können, hat der letzte Wör-

terbucheintrag den Namen **POLYNOM**, der vorletzte Eintrag heißt **VIERTE**, der drittletzte **DRITTE** usw. Wenn FORTH eine Aufgabe zu erledigen hat, dann trifft es sowohl auf Daten als auch auf Wörter. Wir können uns jetzt auch denken, woran FORTH den Unterschied zwischen Daten und Befehlen (Wörtern) erkennen kann: Daten sind nicht im Wörterbuch eingetragen!

Wenn Sie von FORTH die Definition eines neuen Wortes verlangen, dann legt sich das System einen neuen Wörterbucheintrag an. Das Wörterbuch von FORTH warnt Sie, wenn Sie einen Namen doppelt vergeben wollen, läßt aber diese Möglichkeit durchaus zu. Wenn Sie vorsichtig damit umgehen, können doppelte Wörterbucheinträge durchaus verarbeitet werden. Sie können aber auch Probleme mit sich bringen. Wir wollen uns den doppelten Einträgen kurz zuwenden. Nehmen Sie an, daß wir mit dem Block aus Abbildung 2-17 arbeiten, der BLOCK 115 sein soll. Weiterhin wollen wir annehmen, daß unsere Erstfassung des Programms Fehler enthält. Deshalb edieren wir den Block und laden ihn erneut. Jedesmal, wenn wir dies tun, werden aber auch neue Definitionen für die Wörter **ZWEITE, DRITTE, VIERTE** und **POLYNOM** im Wörterbuch abgelegt. Normalerweise kümmert uns das nicht, denn FORTH findet stets zuerst die letzte Definition im Wörterbuch, d.h., die neueste und somit fehlerfreieste. Wir können also die Warnungen über Wortduplikate, die uns FORTH bringt, außer acht lassen und mit der Programmierarbeit weitermachen. Es könnte jedoch ein anderes Problem auftauchen. Wenn unser Programm in Block 115 sehr viele Fehler enthält, kann es nötig sein, diesen Block verhältnismäßig oft zu bearbeiten oder neu zu laden. Das kann dazu führen, daß das Wörterbuch, welches ja nicht nur die FORTH-Wörter, sondern auch ihre Definitionen enthält, unnötig aufgebläht und zu groß wird. Der für das Wörterbuch vorgesehene Speicherplatz reicht dann nicht mehr aus, und es werden Teile des FORTH-Systems überschrieben, was dazu führt, daß sich der Rechner "aufhängt".

Leider schleichen sich beim Programmieren fast immer Fehler ein, und die Situation stellt sich ein, daß ein Teil der im Arbeitsspeicher des Rechners enthaltenen Information nutzlos ist. Es wäre nun gut, "Schrott" aus dem Wörterbuch entfernen und somit Platz für neue, sinnvolle Einträge gewinnen zu können. Dafür gibt es ein spezielles FORTH-Wort, nämlich **FORGET**. Dies leitet sich her vom Englischen "to forget" = vergessen. Nehmen wir einmal an, daß der BLOCK 115 (vgl. Abbildung 2-17) geladen wurde. Wenn wir

```
FORGET POLYNOM                                        (2-40)
```

eingeben, dann wird das Wort **POLYNOM** aus dem Wörterbuch entfernt, und der Platz, den es und seine Definition im Arbeitsspeicher eingenommen haben, wird freigegeben.

Das Wort **FORGET** kann sogar noch mehr: Wenn wir FORTH dazu veranlassen, ein Wort mittels **FORGET** zu vergessen, dann werden auch all die Wörter mit vergessen, die nach dem betreffenden Wort definiert wurden. Wenn wir also z.B. den Block aus Abbildung 2-17 laden und anschließend eingeben

```
FORGET DRITTE                                    (2-41)
```

dann werden gleichzeitig die Wörter **DRITTE, VIERTE** und **POLYNOM** aus dem Wörterbuch entfernt. Wenn Sie also einen Block für die Fehlersuche laden, dann ist es am besten, wenn Sie bei jedem erneuten Ladevorgang dieses Blockes erst einmal die darin enthaltenen Wörter vergessen lassen. Dies kann man ganz einfach dadurch erreichen, daß man in den Block zwei weitere Einträge mit aufnimmt. Betrachten Sie dazu einmal die Abbildung 2-19, welche eine leichte Modifikation der Abbildung 2-17 darstellt.

```
 0 ( ZWEITE, DRITTE, VIERTE und POLYNOM ) FORGET FOO : FOO ;
 1 : ZWEITE DUP  *  ;
 2 : DRITTE DUP   ZWEITE  *  ;
 3 : VIERTE  DUP  DRITTE  *  ;
 4 :
 5 : POLYNOM      DUP DUP DUP
 6 :              VIERTE 3 *
 7 :              SWAP DRITTE 2 * +
 8 :              SWAP ZWEITE 5 * +
 9 :              SWAP 2 * +
10 :              4 + , ;
11
12
13
14
15
```

ABBILDUNG 2-19: Erweiterte Fassung der Abbildung 2-17, die dafür sorgt, daß zuvor geladene Versionen des Blockes automatisch vergessen werden; nützlich für Programmentwicklung.

Die ersten beiden Instruktionen im Block lauten nun folgendermaßen

FORGET FOO (2-42a)

: FOO ; (2-42b)

Die Anweisung (2-42b) definiert ein Wort mit dem Namen **FOO**, das aber nichts tut. Dennoch macht FORTH pflichtschuldigst einen Wörterbucheintrag für dieses seltsame Wort **FOO** und merkt sich ein paar sehr einfache Instruktionen im Zusammenhang mit **FOO**, die dafür sorgen, daß **FOO** beim Aufruf keinerlei Wirkung zeigt. Die erste Instruktion im Block ist aber (2-42a); wenn wir nun den Block aus Abbildung 2-19 erneut laden, er sich also bereits einmal im Wörterbuch befindet, dann sorgt dieses **FORGET FOO** dafür, daß die letzte Kopie des Blockes aus dem Wörterbuch gelöscht wird. Beim ersten Ladeversuch des Blockes erhalten Sie jedoch eine Fehlermeldung, da sich zu diesem Zeitpunkt noch kein Wort mit dem Namen **FOO** im Wörterbuch befindet. Sie müssen deshalb für das erstmalige Laden des Blockes folgendes eingeben:

: FOO ; (RETURN) (2-43)

Laden Sie jetzt den Block. **FOO** wird sofort vergessen und an seiner Stelle ein neues Wort **FOO** ins Wörterbuch eingetragen. Beim nächsten Laden des Blockes sorgt die Instruktion **FORGET FOO** dafür, daß die alten Einträge in diesem Block aus dem Wörterbuch gelöscht werden. Auf diese Art umgehen wir das Problem, daß bei jedem Neubearbeiten und Neuladen eines Blockes das Wörterbuch mit unnötigem Ballast aufgebläht wird und unser Arbeitsspeicherbereich eingeengt wird. Denken Sie daran, daß Sie (2-43) nur einmal eingeben müssen, und zwar, wenn Sie diesen Block das erstemal laden. Wenn das Programm fehlerfrei ist, dann sollten Sie den Block edieren und die Einträge **FORGET FOO** UND : **FOO ;** aus dem Block entfernen.

Es gibt noch andere Einsatzgebiete für das Wort **FORGET**. Bei der Arbeit mit kleinen Computern wird ein Programm, das gerade in Entwicklung ist, manchmal so groß, daß es nicht mehr in den Arbeitsspeicher paßt und deshalb nicht mehr ausgeführt werden kann.

Bei den meisten Programmiersprachen müssen Sie in diesen Fällen Ihr Programm verkürzen oder die Datenmengen einschränken, mit denen es arbeiten kann. FORTH bietet hier eine andere Alternative. Sie können Ihr Programm so schreiben, daß bestimmte Wörter nur am Anfang des Programms benötigt werden, am Ende aber nicht mehr vorkommen. In diesem Fall brauchen Sie am Anfang des Programms nur diejenigen Wörter zu laden, die an dieser Stelle benötigt werden. Der Speicherplatzbedarf wird so reduziert. Die zu Beginn des Programmlaufs geladenen Wörter werden nur zur Berechnung der ersten Zwischenergebnisse herangezogen. Ist dies geschehen, so können Sie mittels **FORGET** unnötige Wörter aus dem Wörterbuch und somit dem Arbeitsspeicher des Computers löschen, wodurch wieder mehr Platz im Computer zur Verfügung steht. Daraufhin kann der Rest des Programms geladen werden und seine Arbeit beginnen.

All dies können bequemerweise Instruktionen übernehmen, die in Ihrem FORTH-Programm selbst eingestreut sind, so daß sich der Benutzer, der Ihr Programm anwendet, um diese Details gar nicht zu kümmern braucht. Die Vorgehensweise ist dabei etwa folgendermaßen: Sie laden einen bestimmten Block, der die für die ersten Berechnungen benötigten Wörter enthält. Die letzten beiden Befehle in diesem Block sind dann ein **FORGET** sowie ein **LOAD**, das den nächsten Block in den Arbeitsspeicher holt. Die Kombination aus diesen beiden Befehlen sorgt dafür, daß die alten Wörter gelöscht werden und der neue Block geladen wird. In gewisser Weise benutzt diese Technik den Disketten- oder Kassettenspeicher als eine Erweiterung des Hauptspeichers.

Die Methode, in einem Programm immer nur die gerade benötigten Teile in den Hauptspeicher "hereinzuladen", trägt den Namen Overlay-Technik. Dieser Fachausdruck leitet sich vom Englischen "Overlay" = Überlagern her. Bei Anwendung der Overlay-Technik stellen Disketten- oder Kassettenspeicher einen sogenannten virtuellen Speicher dar. Diese Speichermedien übernehmen nämlich bei der Overlay-Technik die Rolle einer gedachten Hauptspeichererweiterung. Overlay-Technik ist gerade bei kleineren Mikrocomputern sehr nützlich, in anderen Programmiersprachen aber schwer zu handhaben. Bei einer herkömmlichen Programmiersprache müßten Sie tatsächlich eine Folge von kleineren, unabhängigen Programmen schreiben, diese einzeln hintereinander laufen lassen und über Dateien die Zwischendaten von einem zum anderen Programm übergeben. Das Aufrufen der einzelnen Dateien und das Übergeben der

Zwischendaten ist in diesem Fall Sache des Benutzers, der also mit zusätzlichem Arbeitsaufwand belastet wird.

Kehren wir noch einmal zu dem Problem der doppelten Namen im FORTH-Wörterbuch zurück. Betrachten Sie einmal das FORTH- Programm in Abbildung 2-20. Hier definieren wir zwei Wörter mit dem Namen **TEST1**.

```
 0 ( Ein Beispiel für Namensduplikate )
 1 : TEST1  2 .  ;
 2 : RUNNER1  TEST1  3 .  TEST1  ;
 3 : TEST1  4 . ;
 4 : RUNNER2  TEST1  TEST1  RUNNER1  TEST1  TEST1  ;
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

ABBILDUNG 2-20: Ein Beispiel für doppelte Namen

Wir haben uns hier auf ganz einfache FORTH-Wörter beschränkt. Zeile 1 definiert das Wort **TEST1**, welches lediglich die Zahl 2 ausdruckt. In Zeile 2 definieren wir **RUNNER1**, welches das Wort **TEST1** aufruft, dann eine 3 ausgibt und erneut **TEST1** aufruft. Zeile 3 enthält die Definition eines weiteren Wortes, das ebenfalls den Namen **TEST1** trägt und die Zahl 4 ausgibt. Wir können also erkennen, welches **TEST1** gerade ausgeführt wird, indem wir nachsehen, ob eine 2 oder eine 4 gedruckt wird. Schließlich definieren wir noch in Zeile 4 das Wort **RUNNER2**, das zweimal **TEST1** aufruft, dann das Wort **RUNNER1** enthält und schließlich noch zweimal **TEST1** ruft. Beim Laden des Blockes in Abbildung 2-20 wird als erstes die Zeile 1 geladen, dann die Zeile 2 usw. Wenn wir also eingeben:

TEST1 (RETURN) (2-44)

dann erhalten wir als Ergebnis eine 4. Das am weitesten hinten stehende Wort im FORTH-Wörterbuch mit dem Namen **TEST1** ist nämlich unsere Definition aus Zeile 3 der Abbildung 2-20. Da das Wörterbuch von hinten durchsucht wird, findet FORTH diese Definition als erste und wendet sie an. Jetzt wollen wir einmal sehen, was bei Ausführung von **RUNNER2** passiert. Wir erhalten folgendes Ergebnis:

4 4 2 3 2 4 4 (2-45)

Wird **RUNNER1** als Teil von **RUNNER2** aufgerufen, dann ruft dieses Wort seinerseits **TEST1**. In diesem Fall wird aber das erste **TEST1** ausgeführt. Diese Version bringt eine 2 auf den Bildschirm. Während der Ausführung von **RUNNER1** beginnt also jede Wörterbuchsuche mit dem ersten Wort, das unmittelbar <u>vor</u> der Compilierung von **RUNNER1** definiert wurde. Das ist der Grund, weswegen jetzt das erste **TEST1** zur Ausführung gelangt. Wenn andererseits **RUNNER2** ausgeführt wird, dann ruft dieses Wort nach Ausführung von **RUNNER1** seinerseits wieder **TEST1**. Diesmal wird jedoch zweimal die 4 ausgegeben. Nachdem nämlich die **RUNNER1** abgearbeitet ist, beginnt die Wörterbuchsuche wieder bei dem letzten Wort, das unmittelbar vor **RUNNER2** definiert wurde, da FORTH ja gerade mitten in der Compilierung und Ausführung von **RUNNER2** steckt. Wenn wir jetzt eingeben: **FORGET TEST1**, dann ist dies genauso, als würde der Block nur die Zeilen 0 bis 2 der Abbildung 2-20 enthalten. In diesem Fall würde bei Aufruf des Wortes **TEST1** eine 2 ausgegeben werden.

Noch eine letzte Warnung: Das Wort **FORGET** läßt sich auf <u>jeden</u> Wörterbucheintrag anwenden. Sie können also sogar den Befehl geben: **FORGET** +! Seien Sie aber vorsichtig mit diesen Dingen, da hierbei FORTH einen Großteil seiner wichtigsten Rechenfähigkeiten verlieren und deshalb arbeitsunfähig werden kann. Dann hilft Ihnen nur noch ein Systemstart weiter.

2.5 Weitere Kommandos zur Stack-Manipulation

Dieser Abschnitt führt einige weitere Kommandos ein, mit denen der Stack manipuliert werden kann und die gelegentlich sehr nützlich sind. Die Kommandos sind den im Kapitel 2-2 eingeführten sehr ähnlich, operieren aber mit zwei Stack-Positionen gleichzeitig. Im Kapitel 2-1 haben wir erfahren, daß in FORTH verarbeitbare Zahlen zwischen -32768 und 32767 liegen müssen. In der Regel werden Integers - so nennt man Zahlen aus diesem Bereich - in zwei aufeinanderfolgenden Speicherwörtern des Computers abgelegt, von denen ein jedes 8 Bit umfaßt. Dabei gehen wir davon aus, daß Ihr Computer Speicherwörter von 8 Bit Länge verwendet. In diesem Fall arbeitet FORTH automatisch mit zwei Speicherwörtern, die also, was den Programmierer betrifft, in FORTH als eine Einheit erscheinen. Wenn wir bisher über eine Integer auf dem Stack gesprochen haben, dann war dieser Stack-Eintrag eine Einheit, obwohl er maschinenintern aus zwei 8-Bit-Bytes (oder Maschinenworten) zusammengesetzt ist. Manchmal ist es nötig, mit Zahlen zu arbeiten, deren Betrag die Schranke 32767 überschreitet. FORTH stellt üblicherweise Möglichkeiten für die Arbeit mit wesentlich größeren Integers zur Verfügung. Solche Zahlen tragen den Namen "doppelt genaue Integer". Dieser Zahlentyp beansprucht jedoch mehr Platz als die normalen, einfach genauen Stack-Einträge. Sie werden deshalb auf zwei hintereinander folgenden Stack-Positionen gespeichert. Wenn wir den Stack manipulieren wollen, darauf aber doppelt genaue Zahlen abgelegt sind, dann brauchen wir spezielle Befehle, die Paare von Stack-Positionen manipulieren. Um etwa eine doppelt genaue Integer mittels **DROP** vom Stack zu entfernen, müßten wir das Wort **DROP** zweimal ausführen.

Das Rechnen mit doppelt genauen Integers erläutern wir noch einmal ausführlich in Kapitel 5. Hier wollen wir uns nur mit den Befehlen beschäftigen, die zur Manipulation eines Stacks mit doppelt genauen Einträgen benötigt werden. Diese Wörter können wir aber nicht nur bei doppelt genauen Integers, sondern auch bei einfach genauen Einträgen benutzen. Sie können nämlich genausogut Paare von einfach genauen Zahlen manipulieren, wie man sie zur Bearbeitung eines doppelt genauen Eintrags benutzen kann. In der Regel gehören die hier besprochenen Befehle nicht zu dem FORTH-Kern, der automatisch geladen wird; es handelt sich vielmehr um Erweiterungswörter, die Sie vermutlich selbst in den Speicher bringen müssen.

2DUP - Dieses Wort dupliziert die obersten beiden Positionen des Stacks, also entweder die obersten beiden Integers oder die oberste doppelt genaue Zahl. **2DUP** ist charakterisiert durch die Stack-Relation

$$n_1\ n_2 \text{ --> } n_1\ n_2\ n_1\ n_2 \qquad (2\text{-}46)$$

Wie Sie bereits wissen, steht in diesen Stack-Relationen "n" für eine einfach genaue ganze Zahl (Integer). Wir wollen in Zukunft doppelt genaue Zahlen mit dem Buchstaben "d" abkürzen und können dann die Stack-Relation für das Wort **2DUP** folgendermaßen formulieren:

d --> d d (2-47)

Ein genauer Vergleich von (2-47) mit (2-16) zeigt, daß diese Stack-Relationen denselben Sachverhalt zum Ausdruck bringen, außer, daß hier zwei Integers "n" immer durch eine doppelt genaue Zahl "d" ersetzt wurden. Wenn wir wollen, können wir uns natürlich eine eigene Definition für **2DUP** schreiben; diese sieht so aus:

```
: 2DUP DUP 3 PICK SWAP ;                    (2-48)
```

Abbildung 2-21 stellt ein Stack-Diagramm für folgendes Programm dar:

```
4 2 2DUP                                    (2-49)
```

2DROP - Das FORTH-Wort **2DROP** entfernt die beiden obersten Integers (bzw. die oberste doppelt genaue Zahl) vom Stack. Seine Stack-Relation ist

$$n_1\ n_2 \text{ -->} \qquad (2\text{-}50)$$

bzw. mit doppelt genauen Zahlen

d_1 --> (2-51)

```
I-----I   I-----I   I-----I   I-----I   I-----I
I  2  I   I  2  I   I  3  I   I  4  I   I  2  I
I-----I   I-----I   I-----I   I-----I   I-----I
I  4  I   I  2  I   I  2  I   I  2  I   I  4  I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I  4  I   I  2  I   I  2  I   I  2  I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I  4  I   I  4  I   I  4  I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I   I     I
I     I   I     I   I     I   I     I   I     I

  a)        b)        c)        d)        e)
```

ABBILDUNG 2-21: Stack-Diagramm für das Programm (2-48). a) Nach Eingeben von 4 und 2 in dieser Reihenfolge; b) nach Ausführung von **DUP**; c) nachdem 3 auf den Stack gepusht wurde; d) nach Ausführung von **PICK**; e) nach Ausführung von **SWAP**.

2SWAP - Mit dem FORTH-Kommando **2SWAP** werden die beiden obersten Stack-Elemente miteinander vertauscht. Dies kann man sich an folgender Stack-Relation deutlich machen:

n_1 n_2 n_3 n_4 --> n_3 n_4 n_1 n_2 (2-52)

Eine eigene Definition von **2SWAP** könnte folgendermaßen aussehen:

: 2SWAP ROT 4 ROLL SWAP (2-54)

Die Abbildung 2-22 ist ein Stack-Diagramm zum folgenden kleinen Programm:

```
8 17 9 7 2SWAP                                          (2-55)
```

```
I-----I   I-----I   I-----I   I-----I   I-----I
I  7  I   I 17  I   I  4  I   I  8  I   I 17  I
I-----I   I-----I   I-----I   I-----I   I-----I
I  9  I   I  7  I   I 17  I   I 17  I   I  8  I
I-----I   I-----I   I-----I   I-----I   I-----I
I 17  I   I  9  I   I  7  I   I  7  I   I  7  I
I-----I   I-----I   I-----I   I-----I   I-----I
I  8  I   I  8  I   I  9  I   I  9  I   I  9  I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I  8  I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I     I   I     I   I     I   I     I   I     I
I-----I   I-----I   I-----I   I-----I   I-----I
I  .  I   I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I   I  .  I
I  .  I   I  .  I   I  .  I   I  .  I   I  .  I

   a)        b)        c)        d)        e)
```

ABBILDUNG 2-22: Stack-Diagramm für Programmbeispiel 2-55. a) nach Eingeben von 8, 17, 9 und 7 in dieser Reihenfolge; b) nach Ausführung von **ROT**; c) nachdem 4 auf den Stack gepusht wurde; d) nach Ausführung von **ROLL**; e) nach Ausführung von **SWAP**.

In der Regel werden die mit Ihrem FORTH-System mitgelieferten Wörter etwas schneller laufen als die selbstdefinierten, die wir hier als Beispiel angegeben haben.

2ROT - Mit dem FORTH-Wort **2ROT** wird die dritte doppelt genaue Integer an die oberste Position des Stacks gebracht. Für einfach genaue Integers sieht die Stack-Relation folgendermaßen aus:

$$n_1\ n_2\ n_3\ n_4\ n_5\ n_6 \ \text{-->}\ n_3\ n_4\ n_5\ n_6\ n_1\ n_2 \qquad (2\text{-}56)$$

Bei doppelt genauen Zahlen haben wir

$$d_1\ d_2\ d_3\ \text{-->}\ d_2\ d_3\ d_1 \qquad (2\text{-}57)$$

2OVER - Das FORTH-Wort **2OVER** dupliziert die zweite doppelt genaue Zahl als oberstes Stack-Element. Für einfach genaue Integers sieht die Stack-Relation so aus:

$$n_1\ n_2\ n_3\ n_4\ \text{-->}\ n_1\ n_2\ n_3\ n_4\ n_1\ n_2 \qquad (2\text{-}58)$$

Bei doppelt genauen Integers haben wir

$$d_1\ d_2\ \text{-->}\ d_1\ d_2\ d_1 \qquad (2\text{-}59)$$

Alle bisher eingeführten Wörter zur Stack-Manipulation sind also völlig analog zu denen, die wir in Abschnitt 2-2 bereits für einfach genaue Zahlen kennengelernt haben.

2.6 Der Return-Stack

Wir haben in den bisherigen Darstellungen stets ohne weitere Unterscheidung von "dem Stack" gesprochen; genaugenommen müßten wir jedoch vom Parameter-Stack bzw. Daten-Stack sprechen, da dieser Stack zur Speicherung der Daten oder Parameter dient, mit denen die einzelnen Wörter arbeiten. Die meisten FORTH-Programmierer sprechen aber nur kurz vom "Stack". Es gibt jedoch noch einen zweiten Stack, den sog. Return-Stack (wörtlich "Rückkehr-Stack"). Diesen Stack benötigt das FORTH-System zur Buchführung über die eigenen Operationen. Falls der Return-Stack durcheinanderkommt, brechen alle Berechnungen ab, und Sie müssen Ihr System neu starten, wobei Sie möglicherweise alle Daten verlieren. Es gibt einige FORTH-Wörter, mit denen Sie den Return-Stack manipulieren können. Es scheint so, als ob Sie für diese Wörter gar keinen Bedarf hätten, da ja immer die Gefahr besteht, das FORTH-System zu zerstören. Befolgen Sie aber bei der Arbeit mit dem

Return-Stack einige einfache Regeln, dann können Sie diesen ohne Gefahr für Ihre eigenen Funktionen einsetzen.

Bei der Ausführung eines selbstdefinierten FORTH-Wortes bedient sich FORTH nicht des Return-Stacks. (Wir müssen diese Feststellung später noch etwas einschränken; auf jeden Fall gilt sie für alle bisher definierten Wörter.) Sie können also ebenso den Return-Stack innerhalb eigener Wörter heranziehen, solange Sie nur dafür sorgen, daß der Return-Stack nach Beendigung Ihres selbstdefinierten Wortes wieder in seinem alten Zustand ist.

FORTH-Programmierer setzen den Return-Stack ein, um Zwischenergebnisse bei Berechnungen zu speichern. Angenommen, Sie wollen die 5. Zahl auf dem Stack mit 7 multiplizieren. Dazu könnten Sie mittels **ROLL** die 5. Zahl an die oberste Stack-Position bringen und anschließend mit 7 multiplizieren. Im Anschluß daran müßten Sie jedoch eine Folge von Operationen unternehmen, um das Produkt wieder an die fünfte Stack-Position zu bringen. Einfacher wäre es, die ersten vier Stack-Einträge mittels **POP** zu entfernen und irgendwo zwischenzuspeichern; dadurch käme automatisch der fünfte Stack-Eintrag an oberste Stelle und könnte bequem mit 7 multipliziert werden. Anschließend müßte man die vier soeben entfernten Zahlen vom Zwischenspeicher wegnehmen und der Reihe nach wieder auf den Stack pushen. Dies ist eine viel bequemere Lösung des obigen Problems. Für solche Zwecke - also das Zwischenspeichern von Werten während einer Berechnung - setzt man oftmals den Return-Stack ein. Deshalb betrachten wir jetzt zwei neue FORTH-Wörter, mit denen wir Einträge vom Parameter-Stack (auch Daten-Stack) auf den Return-Stack legen können und umgekehrt. Selbstverständlich funktioniert auch der Return-Stack nach demselben Prinzip wie der Daten-Stack (vgl. Kapitel 1-4), d.h., das zuletzt auf dem Stack abgelegte Element befindet sich zuoberst auf diesem und wird auch als erstes wieder vom Stack entfernt.

>R - Das FORTH-Kommando **>R** poppt das oberste Stack-Element und pusht es auf den Return-Stack. Diese Operation läßt sich durch folgende Stack-Relation ausdrücken:

n --> (2-60)

In Abbildung 2-23 sehen Sie eine Darstellung des Daten-Stacks und des Return-Stacks vor und nach Ausführung von **>R**. Zur Unterschei-

dung zeichnen wir in unseren Diagrammen den Return-Stack mit doppelt durchgezogenen Seitenbegrenzungen.

R> - Das FORTH-Wort **R>** entfernt den obersten Eintrag auf dem Return-Stack und legt ihn auf den Daten-Stack. Die Operation **R>** ist also die "Umkehrung" von **>R**. Die zugehörige Stack-Relation lautet

--> n (2-61)

```
I-----I  II-----II          I-----I  II-----II
I  97 I  II 213 II          I  46 I  II  97 II
I-----I  II-----II          I-----I  II-----II
I  46 I  II  45 II          I  23 I  II 213 II
I-----I  II-----II          I-----I  II-----II
I  23 I  II 918 II          I  17 I  II  45 II
I-----I  II-----II          I-----I  II-----II
I  17 I  II     II          I     I  II 918 II
I-----I  II-----II          I-----I  II-----II
I     I  II     II          I     I  II     II
I-----I  II-----II          I-----I  II-----II
I  .  I  II  .  II          I  .  I  II  .  II
I  .  I  II  .  II          I  .  I  II  .  II
I     I  II     II          I     I  II     II

        a)                          b)
```

ABBILDUNG 2-23: a) Daten-Stack und Return-Stack; b) Stand von Daten-Stack und Return-Stack nach Ausführung von **>R**; der Return-Stack ist mit doppelt durchgezogenen Seitenlinien dargestellt.

Abbildung 2-24 zeigt Ihnen den Daten-Stack und den Return-Stack vor und nach Ausführung von **R>**. Wenn Sie Ihre eigenen FORTH-Wörter definieren, dann müssen Sie darauf achten, daß stets eine gleiche Anzahl von **>R**- und **R>**-Befehlen in diesem Wort enthalten sind. Bei den bisher geschriebenen Programmen ist das sicher nicht sehr schwierig, da sich darin stets ein Verarbeitungsschritt ohne Unterbrechung an den anderen anreiht. In dem nachfolgenden Kapitel werden wir jedoch noch Möglichkeiten kennenlernen, mit denen Programme verzweigen und unterschiedliche Verar-

beitungsschritte auslösen können. In solchen Programmen müssen Sie äußerst genau darauf achten, daß sich die ausgeführten **>R**-Befehle mit den im Programm aufgerufenen **R>**-Wörtern die Waage halten. Weiterhin werden wir in den folgenden Kapiteln Verschachtelungen von Programmen kennenlernen, also innerhalb eines Wortes eingebettete Programmstrukturen. Auch hier muß die Anzahl der Befehle, die auf den Return-Stack schreiben, genau gleich der Anzahl derer sein, die Daten vom Return-Stack entfernen.

```
I-----I   II-----II          I-----I   II-----II
I  46 I   II  97 II          I  97 I   II 213 II
I-----I   II-----II          I-----I   II-----II
I  23 I   II 213 II          I  46 I   II  45 II
I-----I   II-----II          I-----I   II-----II
I  17 I   II  45 II          I  23 I   II 918 II
I-----I   II-----II          I-----I   II-----II
I     I   II 918 II          I     I   II     II
I-----I   II-----II          I-----I   II-----II
I     I   II     II          I     I   II     II
I-----I   II-----II          I-----I   II-----II
I     I   II     II          I     I   II     II
I     I   II     II          I     I   II     II
I     I   II     II          I     I   II     II

        a)                           b)
```

Abbildung 2-24: a) Der Stack und der Return-Stack; b) die beiden Stacks nach Ausführung von **R>**.

Zurück zu unserem Ausgangsproblem: Schreiben wir ein kleines Programm, das die fünfte Zahl auf dem Stack mit 7 multipliziert.

```
: 5MULT7 >R >R >R >R 7 * R> R> R> R> ;                (2-62)
```

Als erstes entfernen wir die obersten vier Zahlen vom Parameter-Stack und pushen sie auf den Return-Stack. Dann legen wir eine 7 auf den Daten-Stack und führen das Wort * aus. Somit ist die ehedem fünfte Zahl auf dem Daten-Stack mit 7 multipliziert worden. Als nächstes holen wir die zwischengespeicherten vier Werte wie-

der vom Return-Stack und legen sie auf den Daten-Stack. Infolgedessen befindet sich der Return-Stack in seinem ursprünglichen Zustand; auch auf dem Daten-Stack ist alles beim alten, außer daß die fünfte Zahl mit 7 multipliziert wurde.

Wir wenden uns nun einigen Befehlen zu, die eine Zahl vom Return-Stack kopieren und auf den Daten-Stack pushen. Diese Instruktionen verändern den Return-Stack nicht, weswegen wir auch nicht die gleichen Vorsichtsmaßnahmen wie bei **R>** und **>R** treffen müssen.

R@ oder **I** - Das Kommando **R@** dupliziert den obersten Eintrag des Return-Stacks auf den Daten-Stack. Es wird also die Zahl, die sich zuoberst auf dem Return-Stack befindet, jetzt auch noch auf den Daten-Stack gepusht. Dadurch ändert sich der Return-Stack nicht. Die zugehörige Stack-Relation sieht folgendermaßen aus:

$$\text{-->}\ n_{ret1} \qquad (2\text{-}63)$$

Die Abbildung 2-25 zeigt das Stack-Diagramm für eine Anwendung von **R@**. Bei einigen FORTH-Systemen, einschließlich dem MMSFORTH, kann man anstelle des Befehls **R@** auch das Wort **I** verwenden. Gemäß den Standards von FORTH-79 sollte man **I** jedoch nur anwenden, um einen Laufindex für eine DO-Schleife zu erhalten. Wir werden uns diesem Thema in Kapitel 4 noch genauer zuwenden. (Beachten Sie, daß in einigen FORTH-Systemen das Wort **R@** unter Umständen gar nicht vorhanden ist; in diesem Fall müssen Sie auf **I** zurückgreifen.)

I' - Das FORTH-Wort **I'** dupliziert die zweite Zahl am Return-Stack auf den Datenstack. Wiederum bleibt der Return-Stack unberührt von der Operation; lediglich der Daten-Stack wird um einen Eintrag erweitert. Dies charakterisiert die folgende Stack-Relation:

$$\text{-->}\ n_{ret2} \qquad (2\text{-}64)$$

Beachten Sie, daß das Wort **I'** kein Teil des FORTH-79-Standards ist. Es ist jedoch im MMSFORTH und in anderen FORTH-Systemen implementiert.

```
I-----I   II-----II                 I-----I   II-----II
I  27 I   II    6 II                I  6  I   II  6   II
I-----I   II-----II                 I-----I   II-----II
I  54 I   II  49 II                 I  27 I   II  49 II
I-----I   II-----II                 I-----I   II-----II
I  37 I   II    5 II                I  54 I   II  5   II
I-----I   II-----II                 I-----I   II-----II
I     I   II  72 II                 I  37 I   II  72 II
I-----I   II-----II                 I-----I   II-----II
I     I   II      II                I     I   II      II
I-----I   II-----II                 I-----I   II-----II
I     I   II      II                I     I   II      II
I     I   II      II                I     I   II      II
I     I   II      II                I     I   II      II

          a)                                  b)
```

ABBILDUNG 2-25: a) Der Stack und der Return-Stack; b) die beiden Stacks nach Ausführung von **R@**.

J - Mittels **J** können Sie die dritte Zahl des Return-Stacks auf den Daten-Stack duplizieren. Das Wort läßt den Return-Stack unverändert. Seine Stack-Relation ist

$$\text{-->}\ n_{ret3} \qquad (2\text{-}65)$$

Auch dies ist kein Wort von FORTH-79, jedoch in MMSFORTH und anderen FORTH-Systemen implementiert. I' und **J** können jedoch in FORTH-79 zusammen mit DO-Schleifen implementiert sein. Mehr darüber erfahren Sie in Kapitel 4.

Wir wollen nun mit Hilfe der Wörter, die wir zuletzt kennengelernt haben, unser Wort POLYNOM (vgl. Abbildung 2-17) noch einmal definieren. Die modifizierte Version sehen Sie in Abbildung 2-26. Wir gehen hier davon aus, daß die Wörter **ZWEITE, DRITTE** und **VIERTE** wie in Abbildung 2-17 zu Teilen Ihres FORTH-Systems gemacht wurden (z.B. durch Laden des entsprechenden Blockes). Wie Sie bereits wissen, können wir das Wort **POLYNOM** für einen Wert von x = 5 anwenden, indem wir eingeben

5 POLYNOMIAL

Betrachten Sie nun die erste Zeile der Abbildung 2-26. Der Wert für x, in diesem Fall 5, wird vom Daten-Stack entfernt und auf den Return-Stack gepusht. Anschließend wird er in Zeile 2 vom Return-Stack dupliziert und befindet sich jetzt wieder auf dem Daten-Stack. Mit diesem Wert 5 arbeitet nun das Wort **VIERTE**, wobei anschließend das Ergebnis von **VIERTE** mit 3 multipliziert wird. Somit befindet sich jetzt $3x^4$ an oberster Stack-Position. Zeile 4 dupliziert die zwischengespeicherte 5 erneut vom Return-Stack auf den Daten-Stack und ruft das Wort **DRITTE**. Danach pushen wir die 2 auf den Stack und führen * aus. Jetzt haben wir bereits $3x^4$ und $2x^3$ auf dem Stack stehen. Wie Sie sehen können, gleicht das Programm in weiten Teilen dem aus Abbildung 2-17. Die meisten Details der Zeilen 4, 5 und 6 entsprechen der alten Version; die Programme unterscheiden sich lediglich in der Art, in der der Parameter 5 den einzelnen Wörtern **VIERTE**, **DRITTE** und **ZWEITE** übergeben wird. In Zeile 6 drucken wir schließlich das gewünschte Ergebnis aus. Ehe wir nun jedoch unser Wort **POLYNOM** verlassen können, müssen wir erst alle Änderungen, die wir am Return-Stack vorgenommen haben, wieder rückgängig machen. Dafür sorgt das **R>** in Zeile 7, denn es entfernt den obersten Wert vom Return-Stack und legt ihn auf den Daten-Stack. Jetzt haben wir wieder denselben Zustand auf dem Return-Stack wie bei Beginn des Programms. Wir könnten jetzt mit der Definition von **POLYNOM** aufhören, würden aber in diesem Fall auf dem Daten-Stack den zuletzt darauf gepushten Wert (in diesem Fall 5) hinterlassen. Dies ist schlechter Stil und führt zu unnötigem Aufblähen des Stacks. Deshalb haben wir in Zeile 7 noch ein zusätzliches **DROP** aufgenommen. Damit verlassen wir die Wortdefinition.

```
0 ( Alternative Fassung von POLYNOM )
1 : POLYNOM      >R
2                R@ VIERTE 3 *
3                R@ DRITTE 2 *  +
4                R@ ZWEITE 5 *  +
5                R@ 2 *  +
6                4 + ,
7                R>    DROP    ;
```

ABBILDUNG 2-26: Modifizierte Version des Wortes **POLYNOM** aus Abbildung 2-17

2.7 Weitere Arithmetik-Befehle

Wir betrachten nun einige weitere FORTH-Wörter, mit denen wir mathematische Operationen ausführen können. Einige einfache Berechnungen werden nämlich sehr häufig gebraucht, weswegen dafür bereits vorgefertigte FORTH-Wörter zur Verfügung stehen.

1+ - Das Wort **1+** entfernt den obersten Stack-Eintrag, erhöht (inkrementiert) seinen Wert um 1 und pusht anschließend diese Summe auf den Stack. Wir erhalten als Stack-Relation:

n --> n+1 (2-66)

Wir können natürlich unser eigenes 1+ schreiben; dies sieht dann folgendermaßen aus:

```
: 1+ 1 + ;
```
(2-67)

Beachten Sie aber, daß das eingebaute Wort **1+** in der Ausführung schneller ist als ein selbstgeschriebenes, da eingebaute Wörter in Maschinensprache geschrieben sind.

1- - Das FORTH-Wort **1-** entfernt die oberste Zahl vom Stack, subtrahiert davon 1 (dekrementiert sie) und pusht das Resultat auf den Stack. Die zugehörige Stack-Relation lautet:

n --> n-1 (2-68)

2+ - Das FORTH-Wort **2+** entfernt die oberste Zahl vom Stack, addiert darauf 2 und legt das Ergebnis als neuen obersten Eintrag auf den Stack. Die Stack-Relation lautet.

n --> n+2 (2-69)

2- - Das FORTH-Wort **2-** entfernt die oberste Zahl auf dem Stack, subtrahiert davon die 2 und pusht die Differenz wieder auf den Stack. Als Stack-Relation haben wir:

n --> n-2 (2-70)

2* - Das FORTH-Wort **2*** entfernt die oberste Zahl vom Stack, multipliziert sie mit 2 und legt das Produkt auf dem Stack ab. Die Stack-Relation lautet

n --> n*2 (2-71)

Dieses Wort ist zwar noch kein Bestandteil von FORTH-79, ist aber in MMSFORTH und in anderen FORTH-Systemen bereits implementiert.

2/ - Das Wort **2/** entfernt die oberste Zahl vom Stack, dividiert sie durch 2 und legt den entstehenden Quotienten auf den Stack. Seine Stack-Relation ist

n --> n/2 (2-72)

Es gibt keine Variante dieses Wortes, die - ähnlich wie **MOD** - den Divisionsrest berechnet. Auch dieses Wort ist noch kein Teil des FORTH-79-Standards, aber bereits in MMSFORTH und anderen Systemen implementiert.

16* - Das FORTH-Kommando **16*** entfernt die oberste Zahl auf dem Stack, multipliziert sie mit 16 und legt das Produkt auf den Stack. Ihre Stack-Relation lautet:

n --> n*16 (2-73)

Dieses Kommando ist kein Teil von FORTH-79, aber bereits in MMS-FORTH und anderen FORTH-Systemen implementiert.

Wenden wir uns nun einigen Wörtern zu, die Zahlenpaare manipulieren. Diese entfernen im allgemeinen die zwei obersten Einträge vom Stack und ersetzen sie durch einen neuen Wert.

MIN - Das FORTH-Wort **MIN** entfernt die zwei obersten Stack-Einträge und ersetzt sie durch den kleineren der beiden (das Minimum). Seine Stack-Relation lautet:

$$n_1\ n_2 \text{ --> } n_{min} \qquad (2\text{-}74)$$

Wir erhalten also mit 2 3 MIN . (RETURN) eine 2 auf dem Bildschirm. Ähnlich ergibt 4 -30 MIN . (RETURN) das Ergebnis -30. In beiden Fällen ist der Stack nach Ausführung des Punktkommandos leer.

MAX - Das Kommando **MAX** entfernt die zwei obersten Zahlen auf dem Stack und legt die größere der beiden dort wieder ab. Seine Stack-Relation lautet:

$$n_1\ n_2 \text{ --> } n_{max} \qquad (2\text{-}75)$$

Die nächsten beiden Wörter arbeiten wieder nur mit der obersten Zahl auf dem Stack. In beiden Fällen wird diese Zahl entfernt und durch eine andere ersetzt.

NEGATE - Das Wort **NEGATE** entfernt die oberste Zahl auf dem Stack, multipliziert sie mit -1 und legt dieses Produkt auf den Stack. Dadurch wird das Vorzeichen der Ausgangszahl vertauscht. Die Stack-Relation lautet:

n --> -n (2-76)

Wenn wir z.B. eingeben

6 NEGATE . (RETURN) (2-77)

wird als Ergebnis -6 ausgedruckt. Nach Ausführung des Punktkommandos ist der Stack wieder leer, vorausgesetzt, er war vor Ausführung dieses Wortes auch leer.

ABS - Das FORTH-Wort **ABS** entfernt den obersten Stack-Eintrag und liefert dessen Absolutwert. Daher leitet sich auch der Name dieses Wortes her. Der Absolutwert wird dann auf den Stack gelegt. Wie Sie wissen, ist der Absolutwert einer positiven Zahl einfach diese Zahl selbst. Den Absolutwert einer negativen Zahl erhält man, indem man diese Zahl mit -1 multipliziert, also positiv macht. Dies wird durch folgende Stack-Relation ausgedrückt:

$$n \text{ --> } |n| \qquad (2\text{-}78)$$

Wenn wir z.B. eingeben

```
-6 ABS    . (ENTER)                                    (2-79)
```

dann wird als Ergebnis 6 ausgedruckt. Danach (nach Ausführung des Punktkommandos) ist der Stack leer, vorausgesetzt, er war auch vor Ausführung dieses Wortes leer.

2.7.1 Zufallszahlen

Manchmal wollen wir auf unserem Rechner ein Zufallsereignis, wie etwa das Würfeln oder das Werfen einer Münze, simulieren. Im Falle des Würfels müßten wir dazu Zahlen zwischen 1 und 6 in zufälliger Reihenfolge erzeugen können. Beim Münzwurf benötigen wir nur Zahlen zwischen 0 und 1 (oder zwei beliebige andere Zahlen), die in zufälliger Reihenfolge auf dem Bildschirm erscheinen. Im allgemeinen kann das Bedürfnis auftauchen, eine zufällige Folge von Zahlen, die sich in einem vorgegebenen Bereich bewegen, vom Computer erzeugen zu lassen. Natürlich gibt es Computerprogramme, die scheinbare Zufallsfolgen von Zahlen erzeugen. Im streng mathematischen Sinn sind diese Folgen nicht zufällig, sie reichen jedoch für die meisten Anwendungsgebiete aus. Man nennt sie aus diesem Grund auch "Pseudozufallszahlen". Eine Möglichkeit zum

Erzeugen von Zufallszahlen ist in FORTH-79 nicht vorgesehen. Einige FORTH-Systeme kennen aber dennoch die dafür nötigen Befehle. Wir wenden uns deshalb den Wörtern zu, die in MMSFORTH zur Erzeugung von Zufallszahlen zur Verfügung stehen.

RND - Das FORTH-Wort **RND** entfernt die oberste Zahl n_1 vom Stack und erzeugt daraufhin eine Pseudozufallszahl n_2, die zwischen 1 und n_1 liegt. Diese Zufallszahl wird dann auf den Stack gelegt. Die Stack-Relation lautet.

n_1 --> $_2$ (2-80)

Wenn wir z.B. eingeben: 24 **RND** (RETURN), dann wird die 24 vom Stack entfernt und dafür eine Zufallszahl zwischen 1 und 24 dort abgelegt. Der Zufallszahlengenerator verwendet eine sogenannte Ursprungszahl, um eine zufällig erscheinende Zahlenfolge zu erzeugen. Wenn wir die Ursprungszahl ändern, dann sorgen wir auch dafür, daß der Generator eine neue Zahlenfolge produziert.

SEED - Um die Ursprungszahl ändern zu können, müssen wir zwei FORTH-Wörter einsetzen: Eines lautet **SEED**, das andere ist das Ausrufezeichen **!**. Wir schreiben sie in der Reihenfolge **SEED !** hin. Achten Sie auf das Leerzeichen. Die Bedeutung des Wortes **!** erläutern wir in Kapitel 6. Für den Augenblick wollen wir die Folge von Wörtern **SEED !** so behandeln, als handle es sich dabei um einen einzigen Befehl. Das FORTH-Wort **SEED !** entfernt die oberste Zahl vom Stack und macht sie zur neuen Ursprungszahl für den Zufallszahlengenerator. Die Stack-Relation für **SEED !** lautet:

n --> (2-81)

Mittels 10 **SEED !**(RETURN) ändern wir die Ursprungszahl auf 10. Wenn wir die Ursprungszahl in zufälliger Folge ändern wollen, dann können wir mittels **RND** eine Zufallszahl auf den Stack legen und diese dann durch das Wort **SEED !** vom Stack entfernen und zur neuen Ursprungszahl machen. Es gibt aber auch ein Kommando, das all diese Schritte ausführt. Bei Eingabe von **RN1** wird die nächste Zufallszahl erzeugt und als neue Ursprungszahl für die Reihe verwendet. Diese Operation hat keinerlei Auswirkungen auf den Stack.

Die Ursprungszahl bestimmt also die Folge von Pseudozufallszahlen, die generiert werden. Bei gleicher Ursprungszahl erhalten wir stets die gleiche Folge. Im allgemeinen ist die Folge der Pseudozufallszahlen sehr lang, aber endlich. Nach einer gewissen - unter Umständen sehr langen Zeit - beginnt sie sich also zu wiederholen. Wenn wir ein Spielprogramm mittels **RND** schreiben, dann spielt dieses Programm deswegen immer auf die gleiche Weise. Deshalb wäre es wünschenswert, die "Einsprungstelle" in die Reihe der Zufallszahlen selbst bestimmen zu können. Dafür gibt es ein spezielles FORTH-Wort.

RANDOMIZE - Das Wort **RANDOMIZE** benötigt keine Zahl auf dem Stack. Jedesmal, wenn Sie es aufrufen, ändert **RANDOMIZE** die Position in der Folge der Zufallszahlen. Wenn Sie also mit **RANDOMIZE** arbeiten, dann beginnt Ihre Zahlenfolge an einer anderen Stelle und erscheint so unterschiedlich.

2.7.2 Ändern der Zahlenbasis

Anwender, die sich um die Details der Maschinensprache nicht zu kümmern brauchen, arbeiten für gewöhnlich im Dezimalsystem. Dieses uns sehr vertraute System verwendet die Ziffern von 0 bis 9 und wird vornehm als "Zahlensystem zur Basis 10" bezeichnet. Im Unterschied dazu verwenden Computer ein "binäres" Zahlensystem, also ein System mit der Basis 2. Alle in Ihrem Computer gespeicherten Zahlen und alle Werte, mit denen er rechnet, sind zur Basis 2. Bei der Ein- oder Ausgabe von Zahlen findet jedoch eine Umwandlung in das Zehnersystem statt, um dem Menschen die Arbeit zu erleichtern. FORTH bietet Ihnen nun die Möglichkeit, diese Umwandlung zu ändern, so daß Sie auch mit anderen Zahlensystemen außer dem Dezimalsystem arbeiten können. Viele Systemprogrammierer verwenden die Basis 8 (das oktale Zahlensystem) oder die Basis 16 (das hexadezimale Dezimalsystem). Sie können Ihrem System sagen, in welcher Basis Sie die Ein- oder Ausgabe von Zahlen wünschen. (Wenn Sie nicht mit anderen Zahlensystemen vertraut sind, dann können Sie diesen Absatz übergehen. Für ein Verständnis von FORTH ist es nicht wichtig, genauere Kenntnisse über Zahlensysteme zu besitzen.) Die im Folgenden vorgestellten Wörter sind bisher noch kein Bestandteil von FORTH-79. Dies kann sich in Zukunft aber durchaus ändern. Auf jeden Fall sind sie in MMSFORTH und anderen FORTH-Systemen implementiert.

HEX - Mit dem Wort **HEX** ändern Sie die Basis für die Ein-/Ausgabe von Zahlen in das Hexadezimalsystem um. **HEX** hat keine Auswirkungen auf den Stack. Es ändert also keine Zahl auf dem Stack oder irgendwo anders in Ihrem Rechner. Die einzigen Auswirkungen, die **HEX** hat, beziehen sich auf Ihre Ein- und Ausgaben. Sie sollten das Wort **HEX** immer allein verwenden, es also nie zum Bestandteil eines anderen Wortes (einer Definition) machen. Kommt es doch einmal in einem FORTH-Wort vor, dann sollte dieses Wort keine Eingabe von numerischen Daten erfordern.

DECIMAL - Das FORTH-Kommando **DECIMAL** funktioniert genauso wie **HEX**, außer daß es die 10 zur neuen Zahlenbasis macht. Beim Starten Ihres FORTH-Systems ist die Zahlenbasis 10 ohnehin voreingestellt.

OCTAL - Das Wort **OCTAL** stellt auf Zahlenein- und ausgabe im Oktalsystem (Basis 8) um; ansonsten funktioniert es genauso wie **HEX**.

BASE - Das Wort **BASE** wird - ähnlich wie **SEED** - nur zusammen mit dem Wort **!** verwendet. Einige FORTH-Systeme, wie z.B. MMSFORTH, ermöglichen es, die Ein-/Ausgabebasis auf andere Werte als 10, 8 oder 16 zu setzen. Diesen Effekt erreicht man durch zwei FORTH-Wörter, nämlich **BASE !**. Mit dieser Wortfolge wird die oberste Zahl vom Stack entfernt und zur Zahlenbasis für alle folgenden Daten einer Ausgabe gemacht. Ihre Stack-Relation lautet:

```
n -->                                                    (2-82)
```

So ist z.B. 16 **BASE !** gleichbedeutend mit **HEX**.

2.8 Übungsaufgaben

In den folgenden Übungsaufgaben sollten Sie soweit wie möglich Programme und benutzerdefinierte FORTH-Wörter auf Ihrem Computer austesten.

2-1 Schreiben Sie ein FORTH-Programm, das folgende Operationen ausführt:

3+4++5-6

2-2 Wiederholen Sie Aufgabe 2-1 mit (3+4+18-25)*5

2-3 Wiederholen Sie Aufgabe 2-1 mit (3+19-4)/6.

2-4 Schreiben Sie ein FORTH-Programm, das den Quotienten und den Rest des folgenden Ausdrucks berechnet:

(3+4+6)/7

2-5 Wiederholen Sie Aufgabe 2-4 mit (3+4*6)/7

2-6 Schreiben Sie ein FORTH-Programm, das die oberste Zahl auf dem Stack mit 5 multipliziert, die Antwort ausdruckt und die ursprüngliche Zahl auf dem Stack hinterläßt.

2-7 Erörtern Sie Einsatzmöglichkeiten für das Wort **DROP**.

2-8 Wiederholen Sie Aufgabe 2-4 mit dem Ausdruck (4+5-6*3)/7.

Achten Sie darauf, Daten und FORTH-Wörter nicht miteinander zu vermischen.

2-9 Vergleichen Sie die FORTH-Wörter **OVER** und **PICK**.

2-10 Vergleichen Sie die FORTH-Wörter **ROT** und **ROLL**.

2-11 Schreiben Sie ein FORTH-Programm, das die Anzahl der Einträge auf dem Stack ausgibt.

2-12 Legen Sie dar, wie man in FORTH neue Wörter definieren kann.

2-13 Schreiben Sie ein eigenes FORTH-Wort, das die obersten vier Zahlen auf dem Stack addiert und dann das Ergebnis druckt.

2-14 Wiederholen Sie Aufgabe 2-13; diesmal soll der Stack nach Beendigung des Wortes unverändert bleiben.

2-15 Schreiben Sie ein FORTH-Wort, das den Durchschnitt der ersten fünf Zahlen auf dem Stack berechnet. Geben Sie sowohl den Quotienten als auch den Divisionsrest aus.

2-16 Schreiben Sie ein FORTH-Programm zur Berechnung von

$6x^5+3x^4+2x^3-21x^2-17x+5$

für verschiedene Werte von x.

2-17 Schreiben Sie ein FORTH-Programm zur Berechnung von ax^3+b für verschiedene Werte von a, b und x.

2-18 Schreiben Sie ein FORTH-Programm zur Berechnung von

ax^3+bx^2+cx+d

für verschiedene Werte von a, b, c, d und x.

2-19 Erörtern Sie das FORTH-Wörterbuch.

2-20 Erörtern Sie das FORTH-Wort **FORGET**.

2-21 Legen Sie dar, wie **FORGET** bei der Fehlersuche einzusetzen ist.

2-22 Was bedeutet der Ausdruck "doppelt genaue Zahl"?

2-23 Vergleichen Sie die FORTH-Kommandos **DUP** und **2DUP**.

2-24 Schreiben Sie ein eigenes FORTH-Wort, das die gleiche Wirkung wie **2DROP** hat.

2-25 Schreiben Sie ein eigenes FORTH-Wort, das die gleiche Wirkung wie **2ROT** hat.

2-26 Schreiben Sie ein eigenes FORTH-Wort, das die gleiche Wirkung wie **2OVER** hat.

2-27 Welche Hauptaufgabe hat der Return-Stack?

2-28 Welche Probleme können sich ergeben, wenn Sie den Return-Stack in selbstdefinierten Wörtern heranziehen? Wie kann man diese Probleme vermeiden?

2-29 Schreiben Sie ein FORTH-Wort, das die ersten fünf Zahlen auf dem Stack multipliziert, den Stack aber ansonsten unverändert läßt. Bedienen Sie sich dazu des Return-Stacks.

2-30 Warum müssen Sie bei der Verwendung von **>R** und **R<** mehr Sorgfalt walten lassen als bei **R@**?

2-31 Erörtern Sie die FORTH-Wörter **I**, **I'** und **J**.

2-32 Wiederholen Sie Aufgabe 2-18, verwenden Sie aber diesmal den Return-Stack.

2-33 Warum ist es sinnvoll, mittels **DROP** Daten vom Stack zu entfernen, die nicht mehr gebraucht werden?

2-34 Schreiben Sie mit den in Abschnitt 2-7 eingeführten FORTH-Wörtern ein neues FORTH-Wort, das 1 auf den obersten Stack-Eintrag addiert und dann das Ergebnis dupliziert.

2-35 Schreiben Sie ein FORTH-Wort, das die kleinste der ersten vier Zahlen auf dem Stack ausgibt.

2-36 Wiederholen Sie Aufgabe 2-35, lassen Sie jedoch den Stack unverändert.

2-37 Wiederholen Sie Aufgabe 2-35, machen Sie diesmal aber die größte Zahl ausfindig.

2-38 Wiederholen Sie die Aufgabe 2-36, machen Sie diesmal aber die größte Zahl ausfindig.

2-39 Wiederholen Sie die Aufgabe 2-35, machen Sie diesmal aber den größten Absolutbetrag ausfindig.

2-40 Wiederholen Sie Aufgabe 2-36, machen Sie diesmal aber den größten Absolutbetrag ausfindig.

2-41 Wiederholen Sie Aufgabe 2-35, machen Sie diesmal aber den kleinsten Absolutbetrag ausfindig.

2-42 Wiederholen Sie Aufgabe 2-36, machen Sie diesmal aber den kleinsten Absolutbetrag ausfindig.

2-43 Vergleichen Sie die FORTH-Wörter **NEGATE** und **ABS**.

2-44 Was ist eine Zufallszahl?

2-45 Was ist eine Ursprungszahl?

2-46 Schreiben Sie ein FORTH-Wort, das eine Zufallszahl zwischen 1 und 55 erzeugt.

2-47 Schreiben Sie ein FORTH-Wort, das eine Dezimalzahl in ihr hexadezimales Äquivalent umwandelt.

2-48 Wiederholen Sie die Aufgabe 2-46, wandeln Sie diesmal aber in eine Oktalzahl um.

2-49 Schreiben Sie ein FORTH-Wort, das fünf Zahlen in hexadezimaler Darstellung addiert und ihre Summe sowohl hexadezimal als auch dezimal ausgibt.

3

Elementare Ein- und Ausgabeoperationen

3 Elementare Ein- und Ausgabeoperationen

Bisher konnten wir nur über das FORTH-Wort . Zahlen auf dem Bildschirm ausgeben. Dieses Kapitel wird Ihre Kenntnisse über die Ein-/Ausgabe von Daten erweitern. Wir werden FORTH-Kommandos kennenlernen, die uns die Ausgabe von Textmaterial (Wörtern) erlauben. Solcher Text kann dazu dienen, die Ergebnisse von Berechnungen zu erläutern und die Bildschirmausgaben lesbarer zu machen. Zusätzlich kann man mit Textausgaben den Benutzer zur Eingabe von gewünschten Werten auffordern, was besonders für die Benutzer eine große Hilfestellung ist, die keine erfahrenen Programmierer sind. Weiterhin werden wir Verfahren kennenlernen, mit denen wir Zahlendaten lesbarer gestalten können.

3.1 Textausgabe

Dieser Abschnitt widmet sich einigen FORTH-Wörtern, mit denen wir Text auf dem Bildschirm des Terminals ausgeben können. Wenn wir etwa ein Programm laufen lassen, das die Summe von vier Zahlen berechnet, dann wäre es wünschenswert, daß dieses Programm den Text "DIE SUMME LAUTET" ausgibt, um sein Ergebnis zu erläutern. Wenn wir ferner ein Programm schreiben, das eine Wertetabelle ausdruckt (z.B. eine Quadrat- und Kubikzahlentabelle), dann wäre es wünschenswert, Überschriften über diese Tabelle drucken zu können.

." und " - mit den FORTH-Wörtern ." und " kann man Textmaterial ausgeben. Wenn wir eingeben:

```
    ." JETZT DRUCKEN WIR TEXT " (RETURN)                (3-1)
```

so bekommen wir auf unserem Bildschirm zu sehen:

```
    JETZT DRUCKEN WIR TEXT ok                           (3-2)
```

Wie Sie sehen können, befindet sich ein Leerzeichen hinter dem Wort .", ebenso, wie sich hinter jedem FORTH-Wort ein Leerzeichen befinden muß. Der gesamte folgende Text hinter dem Leerzeichen wird auf dem Bildschirm ausgegeben. Er endet mit dem Kommando ". Diese beiden Wörter haben keine Auswirkung auf den Stack. Ehe wir uns einem komplizierteren Beispiel für die Ausgabe von Textmaterial zuwenden, wollen wir noch einige weitere FORTH-Wörter kennenlernen.

#IN - Das FORTH-Kommando **#IN** dient der Eingabe von Zahlen. Wenn der FORTH-Interpreter auf **#IN** stößt, dann unterbricht er seine Arbeit und gibt ein Fragezeichen auf Ihrem Bildschirm aus. Anschließend können Sie eine ganze Zahl im Bereich von -32768 bis 32767, gefolgt von RETURN, eingeben. Die eingegebene Zahl wird auf den Stack gelegt, und FORTH fährt mit der Arbeit fort. Das Wort **#IN** ist kein Bestandteil von FORTH 79, findet sich aber in MMSFORTH; die meisten anderen FORTH-Systeme verfügen über ähnliche Wörter.

CR - Das FORTH-Wort **CR** gibt die Zeichen für Wagenrücklauf und Zeilenvorschub auf Ihrer Konsole aus. Dies bewirkt, daß nachfolgendes Textmaterial am Anfang der nächsten Zeile ausgegeben wird. Zwei **CR**-Kommandos hintereinander erzeugen somit eine Leerzeile, mit drei solchen Wörtern erzeugen wir zwei Leerzeilen usw. Auch **CR** hat keine Auswirkungen auf den Stack.

QUIT - Manchmal wollen wir aus ästhetischen Gründen die Ausgabe von "ok" unterbinden, das FORTH nach jedem erfolgreich abgearbeitetem Wort auf den Bildschirm bringt. Dies erreichen wir, wenn das letzte Wort in unserem Programm **QUIT** ist. Das Programm wird beendet, und FORTH ist bereit für die Eingabe neuer Informationen, unterläßt es jedoch, seine Bereitschaftsmeldung "ok" auszugeben. **QUIT** löscht zwar den Return-Stack, läßt aber den Daten-Stack unverändert.

PAGE - In Programmen taucht oft die Notwendigkeit auf, den Bildschirm zu löschen und auf einer neuen, "sauberen" Bildschirmseite mit der Ausgabe zu beginnen. Diese Funktion erfüllt das FORTH-Wort **PAGE**. Es hat keine Auswirkungen auf den Stack. (Auch dieses Wort ist kein Teil von FORTH 79, jedoch in MMSFORTH implementiert.)

Den Einsatz der bisher besprochenen FORTH-Wörter wollen wir an einem Programm verdeutlichen, das die Summe von 4 Zahlen berech-

net. Das Programm unterscheidet sich von den bisher geschriebenen in einem wichtigen Punkt. Wir gehen davon aus, daß es von einem Benutzer bedient wird, der keine Erfahrung im Umgang mit FORTH hat. Deshalb erwarten wir nicht, daß dieser Benutzer erst seine Daten auf den Stack legt, ehe er das Wort aufruft. Stattdessen erfrägt das Programm seine Daten selbst und erklärt das Ergebnis seiner Berechnungen, ehe es dieses auf dem Bildschirm ausgibt. Wir sehen es in Abb. 3-1. Einen Beispiellauf für dieses Programm können Sie der Abb. 3-2 entnehmen.

```
0 ( Beispielprogramm mit Benutzermeldungen und erklärendem Text )
1 : ADD4  PAGE
2        ." Bitte bei jedem Fragezeichen eine Zahl eingeben "
3        CR   ." Eingabe mit RETURN abschliessen "   CR
4        #IN  #IN  #IN  #IN     CR
5        + + +
6        ." Die Summe der vier Zahlen lautet "   CR
7        .    QUIT   ;
8
9
10
11
12
13
14
15
```

ABBILDUNG 3-1: Ein Programm, das im Dialog mit dem Benutzer Daten erfragt und sein Ergebnis beschriftet

```
Bitte bei jedem Fragezeichen eine Zahl eingeben
Eingabe mit RETURN abschliessen
? 4 ? 5 ? 6 ? 7
Die Summe der vier Zahlen lautet
22
```

ABBILDUNG 3-2: Beispiellauf des Programms **ADD4**

Sehen wir uns das Programm einmal genauer an. In Zeile 1 teilen wir dem Compiler mit, welchen Namen wir dem neuen Programm geben wollen. Dieser lautet **ADD4**. Als ersten Schritt in unserem Pro-

gramm löschen wir mittels **PAGE** den Bildschirm. In Zeile 2 sehen wir ein Anwendungsbeispiel für das Textausgabewort **."** und **"**; die beiden Kommandos sorgen dafür, daß der Text "Bitte bei jedem Fragezeichen eine Zahl eingeben" auf Ihrem Bildschirm ausgegeben wird. Das erste Wort in Zeile 3 des Programms, **CR**, sorgt dafür, das der nächste Text auf einer eigenen Zeile beginnt. Auch ihn geben wir mittels **."**, sowie **"** aus, und er lautet "Eingabe mit RETURN abschliessen ". Erneut sorgt ein **CR** dafür, daß nachfolgende Informationen auf eine eigene Zeile zu stehen kommen. Jeder der vier **#IN**-Befehle in Zeile 4 bringt ein Fragezeichen auf den Bildschirm, danach wartet das System so lange, bis der Benutzer mittels RETURN eine Zahl eingegeben hat. Diesen Vorgang können wir der Abbildung 3-2 entnehmen. Die vom Benutzer eingegebenen Zahlen werden in der Reihenfolge auf den Stack gepusht, d.h., die zuletzt eingegebene Zahl befindet sich zuoberst auf dem Stack. Die drei **+**-Wörter in Zeile 5 entfernen nun diese vier Zahlen vom Stack und ersetzen sie durch ihre Summe. Zeile 6 sorgt dafür, daß der Text "Die Summe der vier Zahlen lautet" auf dem Bildschirm erscheint. Die Summe selbst geben wir allerdings auf einer neuen Zeile aus, weswegen als letztes Wort in Zeile 6 noch ein **CR** steht. Die Ausgabe der Summe besorgt in Zeile 7 das Punktkommando. Schließlich beenden wir unser Programm mit **QUIT**, ohne daß bei Beendigung die Bereitschaftsmeldung "ok" ausgegeben wird.

SPACE - Manchmal wollen wir in unseren Text Leerzeichen einfügen. Dies bewerkstelligt das FORTH-Wort **SPACE**. **SPACE** ist also nichts anderes als eine Abkürzung für **."** ". Mit **SPACE** kann man Daten auf dem Bildschirm und im Text positionieren. Wir werden aber noch andere und bequemere Arten zur Formatierung von Ausgabedaten kennenlernen. **SPACE** hat keine Auswirkungen auf den Stack.

SPACES - Das FORTH-Wort **SPACES** entfernt die oberste Zahl vom Stack und gibt eine entsprechende Zahl von Leerzeichen aus. Die Stack-Relation lautet

```
n -->                                                        (3-3)
```

Ein Beispiel:

```
    6 10 5 . SPACES . (RETURN)                               (3-4)
```

Dieses Programm liefert folgendes Ergebnis:

```
5          6 ok                                        (3-5)
```

Wie Sie sehen, stehen zwischen der 5 und der 6 insgesamt 10 Leerzeichen.

PTC - Manchmal wollen wir Informationen an einer bestimmten Stelle am Bildschirm erscheinen lassen. Sicher ist Ihnen schon die Schreibmarke oder der Cursor auf Ihrem Bildschirm aufgefallen; ähnlich wie der Kugelkopf bei einer Kugelkopfschreibmaschine gibt der Cursor an, an welcher Stelle auf dem Bildschirm das nächste Zeichen erscheint, das Sie über Ihre Tastatur eingeben. Mit dem FORTH-Wort **PTC** können Sie diesen Cursor positionieren. Dazu wird der Bildschirm in Zeilen und Spalten unterteilt. **PTC** entfernt die obersten zwei Einträge auf dem Stack. Das oberste Stack-Element gibt die Spalte an, während das nächste Stack-Element die Zeilennummer angibt. Sehen Sie zuerst im Bedienungshandbuch Ihres Computers nach, um herauszufinden, über wieviel Zeilen und Spalten Ihr Bildschirm verfügt. Sie sollten diese Werte in einem Programm auf keinen Fall überschreiten. Die Stack-Relation für das Wort **PTC** lautet

$$n_1 \; n_2 \; \text{-->} \qquad (3\text{-}6)$$

Wenn wir also eingeben:

```
10 6 PTC ." Gruess Gott "                             (3-7)
```

dann wird die Meldung "Gruess Gott" auf Zeile 10 und Spalte 6 ausgegeben. Das Wort **PTC** ist kein Bestandteil von FORTH 79, findet sich aber in MMSFORTH.

3.2 Druckausgabe

Die soeben besprochenen Beispielprogramme konnten ihre Ausgaben nur auf dem Bildschirm machen. Verfügen Sie über einen Drucker, so werden Sie diesen natürlich einsetzen wollen. Zwar ist in FORTH-79 kein spezielles Wort für Druckerausgabe vorgesehen, die meisten FORTH-Systeme verfügen aber über Kommandos, mit denen man Informationen am Drucker ausgeben kann. Wir besprechen hier die einschlägigen Kommandos des MMSFORTH-Systems. Diese Wörter oder ähnlich funktionierende werden wohl auch eines Tages zum Bestandteil von FORTH-79 gemacht.

PRINT - Das Wort **PRINT** sorgt dafür, daß Ausgaben, die normalerweise auf den Bildschirm gehen, auf dem Drucker erfolgen. **PRINT** läßt den Stack unberührt. Bei der Dateneingabe erscheinen die eingegebenen Daten normalerweise auch auf dem Bildschirm. Je nach der Arbeitsweise Ihres Systems kann es sein, daß **PRINT** keinen Einfluß auf die Darstellung von Eingabedaten hat. Diese erscheinen nach wie vor auf dem Bildschirm und gelangen nicht auf den Drucker. Dafür werden alle Programmausgaben auf den Drucker geschickt und sind auf der Konsole nicht mehr zu sehen.

PCRT - Das FORTH-Wort **PCRT** funktioniert genau wie **PRINT**, legt aber Programmausgaben zusätzlich auch auf den Bildschirm. **PCRT** läßt den Stack unverändert. Beachten Sie, daß **PCRT** keine Auswirkung auf die Bildschirmdarstellung hat, sondern lediglich zusätzlich eine Druckausgabe bewirkt.

Sie sollten die Wörter **PRINT** und **PCRT** nicht anwenden, wenn an Ihr System kein Drucker angeschlossen ist. In diesem Fall kann es sein, daß sich Ihr Computer "aufhängt" und Sie gezwungen sind, das System (unter Umständen mit Datenverlust) neu zu starten.

CRT - Mit dem MMSFORTH-Kommando **CRT** wird die Wirkung von **PCRT** und **PRINT** aufgehoben. Nach Ausführung von **CRT** wird die Druckausgabe beendet, und alle Datenausgaben gehen wieder auf den Bildschirm.

Die meisten Kommandos, die wir im ersten Teil des Kapitels besprochen haben, funktionieren auch auf dem Drucker. Man kann also etwa mit **PAGE** den Drucker veranlassen, das Papier auf den Anfang der nächsten Seite vorzuschieben. Die genaue Funktionsweise kann jedoch von System zu System variieren. Spielen Sie deshalb einfach mit diesen Anweisungen etwas herum.

3.3. Der ASCII-Code

In Ihrem Rechner werden alle Informationen in Form von binären Zahlen gespeichert. Dies ist die sog. Interndarstellung. Dennoch kann der Computer auch mit Buchstaben und anderen Symbolen umgehen. Wir wissen ja, daß wir ihm FORTH-Wörter eingeben können und daß er in der Lage ist, Textinformationen auf dem Bildschirm oder Drucker auszugeben. Der Computer kann mit diesen Symbolen arbeiten, weil für jedes eine Zahlenentsprechung, ein sog. Code vereinbart wurde. Die Ein-/Ausgabegeräte Ihres Systems können diese Codes generieren bzw. verstehen. Angenommen, dem Buchstaben A ist die Codezahl 65 zugewiesen. Wenn Sie auf Ihrer Tastatur die Taste mit dem A niederdrücken, dann wird die Zahl 65 in binärer Form an den Computer gesendet. Ähnlich verhält es sich bei der Ausgabe von Informationen: Wenn der Computer ein A ausgeben will, dann gibt er tatsächlich die 65 in binärer Form von sich. Das Terminal bzw. der Drucker decodieren diese Zahl und erzeugen daraus den gewünschten Buchstaben.

Damit Computer mit Terminals und Druckern von verschiedenen Herstellern zusammenarbeiten können, muß man sich auf einen einheitlichen Code festlegen. Es sind zwar mehrere Codevorschriften im Umlauf, die verbreitetste ist jedoch der sogenannte ASCII-Code. Diese Abkürzung kommt von "American Standard Code for Information Interchange". (Amerikanischer Standardcode für Informationsaustausch) Tabelle 3-1 gibt Ihnen einen Überblick über den ASCII-Code.

TABELLE 3-1. ASCII-CODE

Dezimal-wert	Zeichen	Dezimal-wert	Zeichen	Dezimal-wert	Zeichen
0	NUL	47	/	94	^
1	SOH	48	0	95	_
2	STX	49	1	96	[
3	ETX	50	2	97	a
4	EOT	51	3	98	b
5	ENQ	52	4	99	c
6	ACK	52	5	100	d
7	BEL	54	6	101	e
8	BS	55	7	102	f

TABELLE 3-1: ASCII-Code (Forts.)

<table>
<tr><td>9</td><td>HT</td><td>56</td><td>8</td><td>103</td><td>g</td></tr>
<tr><td>10</td><td>LF</td><td>57</td><td>9</td><td>104</td><td>h</td></tr>
<tr><td>11</td><td>VT</td><td>58</td><td>:</td><td>105</td><td>i</td></tr>
<tr><td>12</td><td>FF</td><td>59</td><td>;</td><td>106</td><td>j</td></tr>
<tr><td>13</td><td>CR</td><td>60</td><td><</td><td>107</td><td>k</td></tr>
<tr><td>14</td><td>SO</td><td>61</td><td>=</td><td>108</td><td>l</td></tr>
<tr><td>15</td><td>SI</td><td>62</td><td>></td><td>109</td><td>m</td></tr>
<tr><td>16</td><td>DLE</td><td>63</td><td>?</td><td>110</td><td>n</td></tr>
<tr><td>17</td><td>DC1</td><td>64</td><td>@</td><td>111</td><td>o</td></tr>
<tr><td>18</td><td>DC2</td><td>65</td><td>A</td><td>112</td><td>p</td></tr>
<tr><td>19</td><td>DC3</td><td>66</td><td>B</td><td>113</td><td>q</td></tr>
<tr><td>20</td><td>DC4</td><td>67</td><td>C</td><td>114</td><td>r</td></tr>
<tr><td>21</td><td>NAK</td><td>68</td><td>D</td><td>115</td><td>s</td></tr>
<tr><td>22</td><td>SYN</td><td>69</td><td>E</td><td>116</td><td>t</td></tr>
<tr><td>23</td><td>ETB</td><td>70</td><td>F</td><td>117</td><td>u</td></tr>
<tr><td>24</td><td>CAN</td><td>71</td><td>G</td><td>118</td><td>v</td></tr>
<tr><td>25</td><td>EM</td><td>72</td><td>H</td><td>119</td><td>w</td></tr>
<tr><td>26</td><td>SUB</td><td>73</td><td>I</td><td>120</td><td>x</td></tr>
<tr><td>27</td><td>ESC</td><td>74</td><td>J</td><td>121</td><td>y</td></tr>
<tr><td>28</td><td>FS</td><td>75</td><td>K</td><td>122</td><td>z</td></tr>
<tr><td>29</td><td>GS</td><td>76</td><td>L</td><td>123</td><td>{</td></tr>
<tr><td>30</td><td>RS</td><td>77</td><td>M</td><td>124</td><td>|</td></tr>
<tr><td>31</td><td>US</td><td>78</td><td>N</td><td>125</td><td>}</td></tr>
<tr><td>32</td><td>Leerz.</td><td>79</td><td>O</td><td>126</td><td>~</td></tr>
<tr><td>33</td><td>!</td><td>80</td><td>P</td><td>127</td><td>DEL</td></tr>
<tr><td>34</td><td>"</td><td>81</td><td>Q</td><td></td><td></td></tr>
<tr><td>35</td><td>#</td><td>82</td><td>R</td><td></td><td></td></tr>
<tr><td>36</td><td>$</td><td>83</td><td>S</td><td></td><td></td></tr>
<tr><td>37</td><td>%</td><td>84</td><td>T</td><td></td><td></td></tr>
<tr><td>38</td><td>&</td><td>85</td><td>U</td><td></td><td></td></tr>
<tr><td>39</td><td>`</td><td>86</td><td>V</td><td></td><td></td></tr>
<tr><td>40</td><td>(</td><td>87</td><td>W</td><td></td><td></td></tr>
<tr><td>41</td><td>)</td><td>88</td><td>X</td><td></td><td></td></tr>
<tr><td>42</td><td>*</td><td>89</td><td>Y</td><td></td><td></td></tr>
<tr><td>43</td><td>+</td><td>90</td><td>Z</td><td></td><td></td></tr>
<tr><td>44</td><td>,</td><td>91</td><td>[</td><td></td><td></td></tr>
<tr><td>45</td><td>-</td><td>92</td><td>\</td><td></td><td></td></tr>
<tr><td>46</td><td>.</td><td>93</td><td>]</td><td></td><td></td></tr>
</table>

Wie Sie sehen, gibt es im ASCII-Code auch Einträge, die nicht auf dem Bildschirm darstellbar sind. Hierbei handelt es sich um die ersten 26 Einträge in der ASCII-Tabelle; sie sind für sogenannte Steuerzeichen reserviert. Um ein Steuerzeichen einzugeben, muß man auf den meisten Terminals die CONTROL-Taste niederdrücken und gleichzeitig eine Buchstabentaste anschlagen. Ein gleichzeitiges Drücken der Tasten für CONTROL und A erzeugt dann den ASCII-Code 1, CONTROL-B liefert den ASCII-Code 2 usw. Die ersten 26 ASCII-Zeichen erzeugen bei Ausgabe auf dem Bildschirm keine Zeichen, sondern stellen vielmehr Befehle an den Bildschirm oder andere periphere Geräte dar. So sorgt das ASCII-Zeichen 8 (das man mittels CONTROL-H eingeben kann) dafür, daß sich der Cursor um eine Position nach links auf dem Bildschirm bewegt. Mittels ASCII 10 erzeugt man einen Zeilenvorschub. Das ASCII-Zeichen 13 bewirkt einen Wagenrücklauf am Drucker; auf dem Bildschirm sorgt es dafür, daß der Cursor (ohne Zeilenvorschub) an den Anfang der aktuellen Zeile geht. Das ASCII-Zeichen Nummer 12 bewirkt einen Formularvorschub; es sorgt dafür, daß der Drucker das Papier an den Anfang der nächsten Seite transportiert. (Dies funktioniert natürlich nur, wenn der Drucker auf diese Steuerzeichen auch reagieren kann.)

Mit dem ASCII-Zeichen Nr. 27 hat es eine besondere Bewandtnis. Es ist das sog. Escape-Zeichen. Folgen von ASCII-Zeichen, die mit dem Zeichen Nr. 27 eingeleitet sind, werden auch als Escape-Sequenzen bezeichnet. Diese braucht man oft, um Drucker und andere Peripheriegeräte zu steuern. Viele Graphikdrucker sind über Escape-Sequenzen programmierbar, so daß man per Programm Zeichnungen auf diesen Graphikdruckern erstellen kann. Die ASCII-Codes für die Ziffern 0 bis 9 dürfen Sie nicht mit den Zahlen 0 bis 9 verwechseln; die Ziffer 0 hat vielmehr die Codenummer 48 und die 9 die Nummer 57. Die anderen Ziffernzeichen liegen dazwischen.

Ursprünglich wurde der ASCII-Code für die Arbeit mit Fernschreibern und ähnlichen Geräten entwickelt. Viele der Steuerzeichen wurden zur Steuerung dieser Geräte benötigt. Im nächsten Abschnitt zeigen wir Ihnen, wie man diese Steuerzeichen zusammen mit Ausgabedaten von Computern einsetzen kann.

3.4 Formatierte Zahlen

In diesem Kapitel erfahren Sie, wie man das Aussehen von Zahleninformationen beeinflussen kann, wodurch die Ergebnisse Ihres Computers lesbarer werden. Man nennt diesen Vorgang das Formatieren von Zahlen.

3.4.1 Datenfelder

Man kann für eine Zahl eine bestimmte Anzahl von Stellen auf dem Bildschirm oder auf dem Drucker vorsehen; man spricht in diesem Zusammenhang von einem Datenfeld. FORTH richtet Zahlen in solchen Datenfeldern rechtsbündig aus. Das bedeutet, daß die letzte Ziffer der Zahl am rechten Rand des Feldes zu stehen kommt. Wenn wir ein Datenfeld mit 10 Stellen haben, in das wir eine dreistellige positive Zahl schreiben lassen, dann erscheinen vor dieser Zahl 7 Leerstellen.

.R - Mit dem FORTH-Wort **.R** können wir ein Datenfeld für die Zahlenausgabe festlegen. Das Wort benutzt die obersten zwei Stack-Einträge. Sie werden entfernt, wobei der erste Eintrag die Feldbreite darstellt, der zweite Eintrag die auszugebende Zahl ist. Die Stack-Relation sieht folgendermaßen aus:

$n_1\ n_2$ --> (3-8)

Wenn wir z.B. eingeben

```
23  10  7  10  .R  .R  (RETURN)
```
(3-9)

dann erhalten wir das Ergebnis aus der Abb. 3-3, wobei leere Kästchen in diesem Diagramm für Leerstellen stehen. Sowohl die 23 als auch die 7 werden rechtsbündig in Datenfeldern ausgegeben, die jeweils eine Breite von 10 Stellen haben. Wir können natürlich auch für jede Zahl eine andere Feldbreite angeben.



ABBILDUNG 3-3: Zwei Integers in einem Datenfeld von jeweils 10 Stellen Breite

Durch die Angabe von Datenfeldern können wir die Druckausgabe von Zahlen lesbarer gestalten. Die auszugebenden Daten werden natürlich in den meisten Fällen erst vom Programm berechnet. Man weiß also nicht von vornherein, wie viele Stellen das Ergebnis haben wird. Deshalb kann es vorkommen, daß ein Ergebnis größer ist, als die vorgesehene Feldbreite erlaubt. In diesem Fall vergrößert FORTH automatisch das Datenfeld, so daß die Zahl noch hineinpaßt. Beachten Sie, daß **.R** zur Zeit noch kein Teil von FORTH-79 ist. Es ist jedoch in MMSFORTH und anderen FORTH-Systemen enthalten. Lassen Sie uns als Beispiel ein eigenes Wort schreiben, das Zahlen in ein Datenfeld von 15 Stellen Breite ausgibt:

```
: .15  15  .R ;                                    (3-10)
```

Das Wort hat den Namen **.15**. Es pusht als erstes die 15 auf den Stack; anschließend wird **.R** aufgerufen. Somit wird die Zahl, die vor Ausführung des Wortes **.15** auf dem Stack war, in einem Datenfeld der Breite 15 ausgegeben.

3.4.2 Zahlenausgabe mit Maske

Oft wollen wir beim Ausgeben von Zahlen noch andere Symbole mit in unsere Ausgabe aufnehmen. Bei Geldbeträgen kann es beispielsweise nötig sein, ein Dezimalkomma zu drucken. FORTH läßt auch diese Möglichkeit zu; man bedient sich dazu der sog. **Ausgabemasken.** Zum Einsatz dieser Möglichkeit brauchen wir jedoch mehrere Wörter. Diese werden wir im folgenden Abschnitt darstellen.

Ehe wir uns der Erörterung von Ausgabemasken zuwenden, wollen wir jedoch einen kurzen Exkurs machen und noch einmal über doppelt

genaue Zahlen reden. Diese Zahlen haben wir bereits in Abschnitt 2-5 kennengelernt. Operationen mit diesem Zahlentyp werden wir in einem der folgenden Kapitel besprechen. Hier erörtern wir nur einige grundlegende Fakten, die für den Umgang mit Ausgabemasken notwendig sind. Die elementaren Befehle für die Ausgabe mit Maske arbeiten nämlich nur mit doppelt genauen Integers. (Wir wollen für den Augenblick davon ausgehen, daß diese Zahlen alle positiv sind.) Wir hätten aber ganz gern auch einfach genaue Integers mit einer Druckmaske ausgegeben. Wir wollen einmal sehen, wie dies möglich ist. Eine einfach genaue Zahl benötigt - wie Sie bereits wissen - nur eine Stack-Position, während zur Speicherung von doppelt genauen Zahlen zwei Stack-Einträge benötigt werden. Diesen Sachverhalt illustrieren wir in Abb. 3-4. Wie Sie sehen, haben wir den obersten Stack-Eintrag in Abb. 3-4b mit MSB beschriftet; diese Abkürzung steht für "most signifikant bit" und bezieht sich auf das höchstwertige Bit der Zahl. Entsprechend ist die zweite Stack-Position mit LSB (Abk. für "least signifikant bit", niedrigstwertiges Bit der Zahl) beschriftet.

Bei der Arbeit mit Dezimalzahlen würden wir anstelle von höchstwertigen und niedrigstwertigen Bits von höchstwertigen und niedrigstwertigen Dezimalstellen sprechen. Machen wir uns das am Beispiel der Dezimalzahl 356789 klar. Ihre drei höchstwertigen Ziffern sind 356, während ihre drei niedrigstwertigen 789 sind. Nehmen wir nun an, daß wir die dreistellige Zahl 789 mit 6 Stellen schreiben wollen.

Wir wollen also die Ziffernfolge 000789 ausgeben. In diesem Fall haben wir eine Zahl, deren erste drei höchstwertige Stellen 0 sind. Nun wollen wir sehen, wie wir eine einfach genaue positive Integer als oberstes Stack-Element in eine doppelt genaue umwandeln können. Dies geht ganz einfach dadurch, daß wir eine 0 auf den Stack pushen. Jetzt haben wir eine doppelt genaue Integer, deren höchstwertige Stellen allesamt 0 sind. Von ihrem Betrag her ist diese doppelt genaue Zahl natürlich immer noch die gleiche wie die vorherige einfach genaue Zahl. Das gilt allerdings nur, wenn die Zahl positiv ist.

Die meisten Computer verwenden nämlich ein binäres Zahlensystem, bei dem das höchstwertige Bit (MSB) dazu dient mitzuteilen, ob die Zahl positiv oder negativ ist. Wenn wir jetzt bei einer negativen Zahl einfach 0 auf den Stack pushen, um sie doppelt genau zu machen, dann erhalten wir dadurch die falsche Zahl. Um mit Zahlen zu arbeiten, die sowohl positiv als auch negativ sein kön-

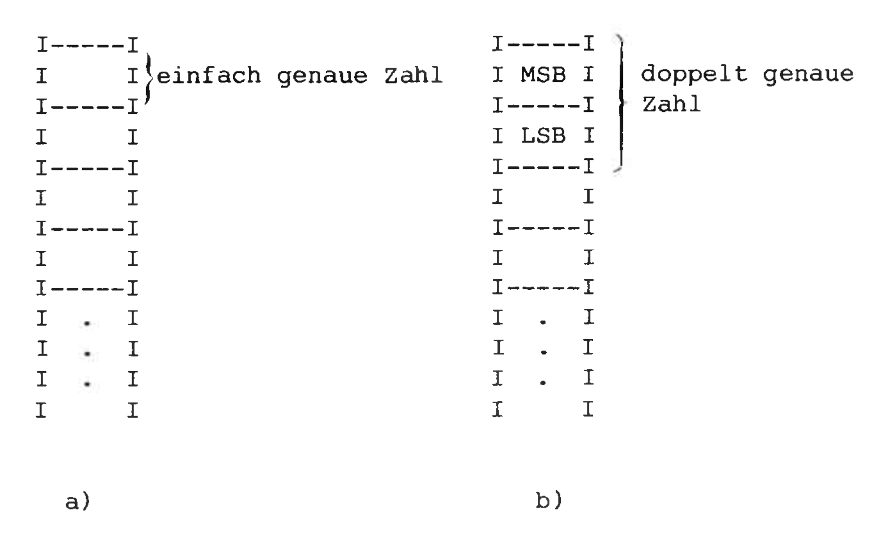


ABBILDUNG 3-4: Der Stack a) mit einer einfach genauen Zahl
b) mit einer doppelt genauen Zahl

nen, müssen wir deshalb das folgende allgemeine Verfahren anwenden. Zuerst duplizieren wir mittels **DUP** die einfach genaue Zahl auf den Stack. Dann ersetzen wir mittels **ABS** diese Zahl durch ihren Absolutbetrag. Wenn wir jetzt eine 0 auf den Stack pushen, dann haben wir den korrekten Absolutbetrag in Form einer doppelt genauen Zahl. Die ursprüngliche einfach genaue Zahl befindet sich immer noch auf dem Stack, und zwar an dritter Position. Wir können sie dazu benutzen, das Vorzeichen der Ausgangszahl zu bestimmen. Wie man diese Prozedur genau in FORTH implementiert, zeigen wir Ihnen noch später in diesem Abschnitt. Erst wollen wir uns einmal den Kommandos für die Ausgabe mit Maske zuwenden.

<# und **#>** - Die FORTH-Kommandos **<#** und **#>** werden für die Ausgabe von Zahlen mit Ausgabemaske verwendet. Ehe wir ihre Anwendung erörtern können, müssen wir jedoch noch ein paar andere Wörter einführen. Der Grundgedanke bei der Verwendung dieser beiden Wörter ist jedoch ganz einfach. Eine Formatanweisung mit diesen beiden Wörtern hat immer folgenden Aufbau:

```
<# Formatanweisungen #>                                    (3-11)
```

Bei Ausführung eines Befehls in dem Format aus Abb. 3-11 wird die doppelt genaue Zahl an oberster Stack-Position vom Stack entfernt. Die Formatanweisungen haben die Form einer Folge von ASCII-Zeichen, eines sog. Zeichenstrings. Jedes Zeichen dieses Strings stellt eine einzelne Ziffer der auszugebenden Zahl oder ein anderes auszugebendes Zeichen dar. Nach Ausführung einer Anweisung von der Form 3-11 finden sich die ASCII-Codes nicht auf dem Stack, sondern in aufeinanderfolgenden Speicherstellen des Arbeitsspeichers. Als letztes pusht eine Formatanweisung von der Form 3-11 zwei einfach genaue Integers auf den Stack. Eine stellt die Startadresse dar, an der sich die soeben gespeicherten ASCII-Zeichen befinden; die zweite gibt an, wie viele Zeichen gespeichert sind (d.h., die Anzahl der Zeichen, die gedruckt werden sollen). Letztere Zahl befindet sich an oberster Stack-Position. Die Stack-Relation für eine Anweisung der Form 3-11 lautet daher

$$n_1 \; n_2 \; \texttt{-->} \; n_{addr} \; n_{länge} \qquad \text{12a)}$$

Dies gilt für einfach genaue Zahlen, wobei n_1 die ursprüngliche einfach genaue Integer und n_2 gleich 0 ist. Bei doppelt genauen Zahlen sieht die Stack-Relation so aus:

$$d \; \texttt{-->} \; n_{addr} \; n_{länge} \qquad \text{(3-12b)}$$

Wenden wir uns nun wieder der Anweisung 3-11 zu. Die Formatanweisung zwischen den beiden FORTH-Wörtern <# und #> werden von links nach rechts durchgegangen. Dabei gilt die erste im String enthaltene Formatanweisung für die letzte Stelle der Zahl. Die zweite Formatanweisung bezieht sich auf die vorletzte Stelle usw. (Die Formatanweisungen müssen sich nicht notwendigerweise nur auf Stellen in der auszugebenden Zahl beziehen. Wir können auch andere Symbole, wie z.B. ein Dezimalkomma, mit aufnehmen.) Jetzt müssen wir nur noch erfahren, welche FORTH-Kommandos innerhalb dieser Formatanweisungen stehen können.

- Mit dem # manipulieren wir einzelne Stellen in einer Zahl. Bei Ausführung einer Anweisung wie in 3-11 wird ja eine doppelt genaue Integer von oberster Stack-Position letztendlich in eine Folge von ASCII-Zeichen umgewandelt, wobei je ein ASCII-Zeichen für eine Ziffer steht. (Wenn FORTH im Dezimalsystem arbeitet,

dann handelt es sich hierbei um Dezimalziffern. Haben wir Basis 16 mittels **HEX** gewählt, dann sind es Hexadezimalziffern usw.) Für die vorliegende Diskussion wollen wir annehmen, daß es sich um Dezimalzahlen handelt. Das Wort **#** (das nur zwischen <**#** und **#**> stehen kann) nimmt die niedrigstwertige Dezimalstelle und entfernt sie von der Zahl. Auch die Tatsache, daß die Zahlen intern im Binärsystem gespeichert sind, soll uns hier nicht stören. Anschließend wird das ASCII-Zeichen für diese Stelle im Arbeitsspeicher abgelegt. Wir können das Wort **#** nur zusammen mit positiven Zahlen verwenden. (Vergleichen Sie die Ausführungen über negative Zahlen am Anfang dieses Abschnitts.) Denken Sie daran, daß bei jeder Ausführung eines **#**-Wortes eine Dezimalstelle von der Zahl entfernt wird.

#S - Das FORTH-Wort **#S** wandelt eine doppelt genaue Zahl in eine Folge von ASCII-Zeichen um. Jedes dieser Zeichen stellt eine Dezimalstelle dieser Zahl dar. Auch **#S** sollte nur mit doppelt genauen positiven Integers verwendet werden. Ist die Zahl 0, dann wird nur eine einzelne Null (als Ziffer) ausgegeben. Ebenso wie bei **#** darf auch **#S** nur zwischen den Wörtern <**#** und **#**> stehen. Beachten Sie, daß **#** nur eine einzige Dezimalstelle umwandelt, während **#S** die ganze verbleibende Zahl in die entsprechende Folge von ASCII-Zeichen konvertiert.

HOLD - Mit dem FORTH-Kommando **HOLD** können wir den ASCII-Code eines beliebigen Symbols in den String von ASCII-Zeichen mit aufnehmen, der unsere Zahl darstellt. **HOLD** entfernt die oberste einfach genaue Integer vom Stack und nimmt das entsprechende Zeichen der ASCII-Tabelle in den Zeichenstring mit auf. Die Stack-Relation des Wortes lautet:

n --> (3-13)

Beachten Sie, daß auch **HOLD** nur zwischen den Wörtern <**#** und **#**> stehen kann.

Als Beispiel für die bisher vorgestellten FORTH-Wörter wollen wir ein neues Wort definieren, das eine einfach genaue positive Integer so ausgibt, daß ein Dezimalkomma vor den letzten beiden Stellen der Zahl steht. Das Einfügen eines Dezimalkommas kann aus mehreren Gründen erwünscht sein. Wenn Sie z.B. Programme schreiben, die mit Geldbeträgen umgehen, dann werden Sie die Mark- und

Pfennigbeträge kenntlich machen wollen. Da wir bisher nur mit Integers arbeiten können, können wir in unseren Programmen auch keine Zahlen mit Nachkommastellen eingeben. Wir müssen dann den Betrag 34,50 DM als die Zahl 3450 eingeben und intern verarbeiten. Beim Ausgeben der Geldbeträge sollte das Dezimalkomma jedoch wieder erscheinen. Unser selbstdefiniertes Wort leistet genau das Gewünschte: Es gibt eine Zahl aus, wobei sich vor den letzten beiden Stellen der Zahl ein Dezimalkomma befindet. Seine Definition:

```
: WRD 0 <# # # 44 HOLD #S #> ;                    (3-14)
```

Sehen wir uns das neue Wort **WRD** einmal genauer an. Wir gehen davon aus, daß sich eine einfach genaue Integer in oberster Stack-Position befindet. Deshalb pushen wir erst einmal die 0 auf den Stack, um daraus eine doppelt genaue Integer mit demselben Betrag zu machen (erinnern Sie sich noch?). Jetzt kommen die Befehle an die Reihe, die zwischen **<#** und **#>** liegen. Das erste **#**entfernt die niedrigstwertige Dezimalstelle von unserer doppelt genauen Zahl auf dem Stack. Daraufhin wird das ASCII-Zeichen generiert, das dieser Zahl entspricht, und im Speicher abgelegt. Das zweite **#** sorgt für dieselbe Prozedur mit der niedrigstwertigen Dezimalstelle der verbleibenden Zahl. Als nächstes legen wir die Zahl 44 auf den Stack. **HOLD** entfernt sie wieder und sorgt dafür, daß im String der ASCII-Zeichen eine 44 mit abgespeichert wird. Der ASCII-Tabelle können Sie entnehmen, daß die 44 der interne Code für das Komma ist.

Wir haben also jetzt in unseren Ergebnisstring das Dezimalkomma geschrieben. Als nächster Befehl ist **#S** an der Reihe. Dieses Wort wandelt die ganze verbleibende doppelt genaue Zahl in einen String von ASCII-Zeichen um, eins für jede Dezimalstelle. Nach Ausführung von **#>** steht also der komplette String im Arbeitsspeicher. Außerdem haben wir ein Dezimalkomma vor die beiden letzten Stellen der Zahl eingefügt. Die Originalzahl ist vom Stack verschwunden. An ihrer Stelle sind zwei neue Zahlen auf den Stack gekommen. Diese stellen die Startadresse des Ergebnisstrings im Arbeitsspeicher dar, den wir eben erzeugt haben; die zweite Zahl gibt an, wieviele Zeichen dieser String enthält. Das Ergebnis steht aber nach wie vor im Arbeitsspeicher, ist aber noch nicht zu sehen. Dafür - für die Anzeige eines mit Formatmaske erzeugten Ziffernstrings - benötigen wir noch ein eigenes Kommando.

TYPE - Mit dem FORTH-Wort **TYPE** werden ASCII-Zeichen ausgegeben, die im Arbeitsspeicher des Computers gespeichert sind. **TYPE** sorgt dafür, daß diese Zeichen am Ausgabegerät (dem Terminal oder dem Drucker) erscheinen. Dazu entfernt es zwei Zahlen vom Stack. Der oberste Stack-Eintrag gibt an, wieviele Zeichen **TYPE** an das Ausgabegerät übertragen soll. Die zweite Zahl auf dem Stack stellt die Anfangsadresse dar, ab der sich der auszugebende String im Arbeitsspeicher des Computers befindet. Die Stack-Relation für **TYPE** lautet:

$n_{addr}\ n_{länge}$ --> (3-15)

Wir können jetzt unser Programm 3-14 so verändern, daß es sein Ergebnis auch noch auf dem Bildschirm ausgibt. Dazu ändern wir lediglich den Namen der Definition und fügen das zusätzliche Wort **TYPE** mit an:

```
: WRTT 0 <# # # 44 HOLD #S #> TYPE ;                (3-16)
```

(Streng genommen ist es gar nicht nötig, der Definition einen neuen Namen zu geben.) Nachdem FORTH die neue Definition 3-16 compiliert hat, können wir eingeben:

```
15756 WRTT (RETURN)                                 (3-17)
```

und erhalten als Ausgabe auf unserem Bildschirm:

```
157,56 ok
```

SIGN - Die bisher geschriebenen Programme funktionieren nur mit positiven Zahlen. Jetzt wollen wir einmal sehen, wie wir Zahlen ausgeben können, die sowohl positiv als auch negativ sein können. Das FORTH-Wort **SIGN**, das nur zwischen den Wörtern **<#** und **#>** stehen kann, erzeugt den ASCII-Code für ein Minuszeichen und nimmt ihn in den ASCII-String mit auf, falls die Zahl an der obersten Stack-Position negativ ist. Ist sie hingegen positiv, dann tut **SIGN** nichts. Jetzt können wir ein Programm schreiben, das ebenso wie die bisherigen eine Zahl mit Dezimalkomma ausgibt, zusätzlich

aber noch ein Minuszeichen vor die Zahl stellt, falls diese negativ ist.

```
: WRTTN DUP ABS 0 <# # # 44 HOLD #S
  ROT SIGN #> TYPE ;                                    (3-18)
```

Sehen wir uns das neue Wort einmal genauer an. **DUP** dupliziert die auszugebende Zahl. Als nächstes wird die Zahl mittels **ABS** durch ihren Absolutwert ersetzt. Wir haben jetzt den Betrag der Originalzahl an oberster Stack-Position, falls die Originalzahl negativ war, und ansonsten die unveränderte Originalzahl. Dann machen wir aus dieser Zahl durch pushen von 0 eine doppelt genaue Zahl. Als nächstes sehen wir den Befehl **<#**. Dieser leitet die Verarbeitung der doppelt genauen Zahl an oberster Stack-Position ein. Die Befehlsfolge **# # 44 HOLD #S** wandelt diese Zahl ebenso wie das bereits bekannte Beispiel 3-16 in eine Folge von ASCII-Zeichen um. Die 0 und der Absolutwert der eingesetzten Zahl befinden sich immer noch auf dem Stack. Jetzt kommt der Befehl **ROT** an die Reihe. Wie Sie sehen, können also auch "normale" FORTH-Wörter zwischen **<#** und **#>** stehen. Dadurch wird der dritte Stack-Eintrag an die oberste Stelle gebracht. Dabei handelt es sich aber um die Originalzahl, die ausgegeben werden soll. **SIGN** entfernt diese Zahl vom Stack. Falls sie negativ ist, dann fügt das Wort **SIGN** den ASCII-Code für das Minuszeichen vor den Ergebnisstring ein. Ist die fragliche Zahl positiv, dann hat **SIGN** keinerlei Auswirkung. Die Umwandlung der Zahl wird wie üblich mit **#>** abgeschlossen. Jetzt befinden sich Startadresse und String-Länge auf dem Stack. **TYPE** holt sich diese beiden Einträge und sorgt dafür, daß die angegebene Anzahl von Zeichen ab der zur Verfügung gestellten Startadresse auf dem Bildschirm ausgegeben wird. Die Befehle zwischen **<#** und **<#** werden also - wie üblich - von links nach rechts abgearbeitet. Das erste ausgeführte Wort ist also **#**, während das letzte **SIGN** ist. Das Kommando **TYPE** gibt den erzeugten String in umgekehrter Reihenfolge aus: das Ergebnis des zuletzt ausgeführten Befehls wird als erstes gedruckt. Deswegen bewirkt das Wort **SIGN**, welches das letzte Kommando zwischen **<#** und **#>** ist, daß ein Minuszeichen vor die Zahl gedruckt wird, vorausgesetzt, diese ist negativ.

Als letztes Beispiel wollen wir (3-18) so verändern, daß die Zahl mit einem Dollarzeichen nach dem Vorzeichen ausgegeben wird. Das veränderte Wort sieht nunmehr folgendermaßen aus:

```
: WRT$ DUP ABS 0 <# # # 44 HOLD #S
  36 HOLD ROT SIGN #> TYPE ;                    (3-19)
```

Im wesentlichen handelt es sich hierbei um die gleiche Definition wie in (3-18), außer daß der Befehl 36 **HOLD** hinzugekommen ist. Wie Sie der ASCII-Tabelle entnehmen können, ist die Code-Zahl des Dollarzeichens 36. Deshalb wird das ASCII-Zeichen in die Zeichenkette mit aufgenommen und taucht dort vor dem Minuszeichen auf.

3.5 Übungsaufgaben

Überprüfen Sie bei den folgenden Aufgaben alle Programme und selbstdefinierten Wörter, indem Sie sie auf Ihrem Computer laufen lassen.

3-1 Was bedeutet der Begriff "Text"?

3-2 Schreiben Sie ein FORTH-Wort, das Ihre Anschrift auf dem Bildschirm ausdruckt.

3-3 Schreiben Sie ein FORTH-Wort, das den Durchschnitt von 4 Zahlen berechnet. Das Wort soll den Benutzer zur Eingabe der Werte auffordern und anschließend warten, bis der Benutzer sie zur Verfügung gestellt hat. Quotient und Divisionsrest des Durchschnitts sollen dann mit entsprechenden Erklärungen auf dem Bildschirm ausgegeben werden.

3-4 Ändern Sie die Definition von Aufgaben 3-3 so ab, daß die Bereitschaftsmeldung "ok" nicht ausgegeben wird.

3-5 Ändern Sie die Definition von Aufgabe 3-4 so ab, daß Quotient und Divisionsrest auf verschiedenen Zeilen ausgegeben werden.

3-6 Ändern Sie mittels **SPACES** die Definition von Aufgabe 3-5 so, daß genügend Zwischenraum zwischen den Ergebnissen steht.

3-7 Wiederholen Sie Aufgabe 3-2, und senden Sie das Ergebnis direkt an Ihren Drucker.

3-8 Wiederholen Sie Aufgabe 3-3, und senden Sie das Ergebnis direkt an Ihren Drucker.

3-9 Wiederholen Sie Aufgabe 3-4, und senden Sie das Ergebnis direkt an Ihren Drucker.

3-10 Wiederholen Sie Aufgabe 3-5, und senden Sie das Ergebnis direkt an Ihren Drucker.

3-11 Wiederholen Sie Aufgabe 3-6, und senden Sie das Ergebnis direkt an Ihren Drucker.

3-12 Was ist ein ASCII-Code?

3-13 Schreiben Sie eine Tabelle mit den ASCII-Codes für die Steuerzeichen.

3-14 Angenommen, der ASCII-Code eines Großbuchstabens befindet sich an oberster Stack-Position. Schreiben Sie ein FORTH-Wort, das diese Zahl in den ASCII-Code für einen entsprechenden Kleinbuchstaben umwandelt.

3-15 Wiederholen Sie Aufgabe 3-14, wandeln Sie diesmal jedoch einen Kleinbuchstaben in einen entsprechenden Großbuchstaben um.

3-16 Was ist ein Datenfeld?

3-17 Schreiben Sie ein FORTH-Wort, das die beiden obersten Stackeinträge ausgibt. Die Zahl an oberster Stack-Position sollte in einem Feld von 15 Zeichen Breite rechtsbündig erscheinen. Die zweite Zahl soll rechtsbündig in einem 25 Zeichen breiten Feld stehen.

3-18 Erörtern Sie das Prinzip der Zahlenausgabe mit Maske.

3-19 Legen Sie dar, wie man eine positive einfach genaue Integer in eine entsprechende doppelt genaue Integer umwandeln kann.

3-20 Schreiben Sie ein FORTH-Wort, das eine fünfstellige Zahl mit einer Leerstelle nach den ersten zwei Ziffern druckt. Die Zahl soll positiv sein.

3-21 Schreiben Sie ein FORTH-Wort, das eine fünfstellige Zahl mit der Beschriftung "DM" nach den ersten drei und mit "Pfennig" nach den letzten beiden Stellen ausgibt. Die Zahl soll positiv sein.

3-22 Kann man eine negative einfach genaue Integer in eine doppelt genaue Integer verwandeln, indem man eine 0 auf den Stack pusht?

3-23 Wiederholen Sie Aufgabe 3-20, gehen Sie diesmal aber nicht davon aus, daß die Zahl positiv ist. Wenn sie negativ ist, dann sollten Sie ein Minuszeichen vor die Zahl drucken.

3-24 Wiederholen Sie Aufgabe 3-21, gehen Sie aber nicht davon aus, daß die Zahl positiv ist. Wenn sie negativ ist, dann sollten Sie ein Minuszeichen vor die Zahl drucken.

3-25 Wiederholen Sie Aufgabe 3-20, schicken Sie das Ergebnis aber direkt an den Drucker.

3-26 Wiederholen Sie Aufgabe 3-21, schicken Sie das Ergebnis aber direkt an den Drucker.

4

Programmsteuerung — Strukturiertes Programmieren

4 Programmsteuerung - Strukturiertes Programmieren

Die bisher betrachteten FORTH-Programme haben einfach einzelne Wörter der Reihe nach ausgeführt. Wesentlich leistungsfähigere Programme kann man schreiben, wenn es möglich ist, in einem Programm verschiedene Zweige zu verfolgen. Bei solchen Programmen entscheidet der Computer aufgrund eines Entscheidungsprozesses selbst, welcher Zweig im Programm verfolgt werden soll. Diesen Entscheidungsprozeß können Sie in Ihrem Programm bestimmen. Die Entscheidung kann etwa davon abhängen, ob eine Zahl positiv oder negativ ist. Möglichkeiten, mit denen solche Verzweigungen erreicht werden können, betrachten wir in diesem Kapitel.

Oft ist es auch wünschenswert, daß ein Programm eine Folge von Arbeitsschritten wiederholt ausführt (sog. Programmschleife). Die Techniken, die dies bewirken, werden wir ebenfalls im vorliegenden Kapitel kennenlernen.

Schließlich gibt es noch Methoden zur Programmierung, die Programme mit Verzweigungen und Schleifen weniger fehleranfällig und leichter korrigierbar machen. Die hierzu eingesetzte Methode trägt den Namen "Strukturierte Programmierung" und wird ebenfalls Gegenstand dieses Kapitels sein.

4.1 Logische Bedingungen

In FORTH werden - ebenso wie in anderen Programmiersprachen - Entscheidungen in Abhängigkeit davon getroffen, ob eine Variable den Wert "wahr" oder "falsch" hat. Wir können beispielsweise überprüfen, ob die oberste Zahl auf dem Stack positiv ist. Nach dieser Überprüfung wird die Zahl vom Stack entfernt und darauf entweder eine Eins abgelegt, falls die Zahl positiv war, oder im Falle eines negativen Eintrags eine Null. Diese Eins bzw. Null bezeichnet man auch als Flag. Die Eins steht für den Wahrheitswert "wahr", während die Null "falsch" darstellt. Die Flag-Werte 0 und 1 sind gewöhnliche einfach genaue Integers. Es gibt also keine Möglichkeit, den Flag-Wert 1 von einer Eins zu unterscheiden, die Sie auf dem Stack abgelegt haben. Spätere Abschnitte dieses Kapitels werden sich der Frage widmen, wie man in einem Programm Verzweigungen veranlassen kann, je nachdem, welchen Wert

ein Flag an oberster Stack-Position hat. Dieser Abschnitt führt erst einmal die FORTH-Wörter ein, die für solche Vergleiche benötigt werden, die als Ergebnisse Flags auf den Stack pushen.

0= - Das FORTH-Wort **0=** überprüft auf Gleichheit mit 0. Dazu entfernt es den obersten Stack-Eintrag und legt statt dessen ein Flag darauf ab. Dieses besitzt den Wert 1, falls die Testzahl gleich Null war. In allen anderen Fällen, d.h., wenn zuvor keine Null auf dem Stack war, hat das Flag den Wert 0. Hier die Stack-Relation:

$$n \text{ --> } n_{flag} \qquad (4\text{-}1)$$

Das Flag ist "wahr", falls n = 0. Denken Sie daran, daß das Flag nichts anderes als eine einfach genaue Integer ist. Der Zahlenwert 0 steht dabei für den Wahrheitswert "falsch", während 1 für "wahr" steht. Im Rest des Buches werden wir deshalb einfach sagen, daß ein Flag "wahr" oder "falsch" ist. Abbildung 4-1 zeigt ein Stack-Diagramm vor und nach Ausführung von **0=**. Wie Sie sehen, wurde der oberste Stack-Eintrag, der ungleich 0 ist, entfernt und statt dessen eine 0 (für "falsch") auf den Stack gepusht.

```
I-----I   I-----I
I  7  I   I  0  I
I-----I   I-----I
I 15  I   I 15  I
I-----I   I-----I
I  8  I   I  8  I
I-----I   I-----I
I     I   I     I
I-----I   I-----I
I  .  I   I  .  I
I  .  I   I  .  I
I  .  I   I  .  I
I     I   I     I

   a)        b)
```

ABBILDUNG 4-1: a) Typischer Stack-Zustand;
b) der Stack nach Ausführung von **0=**.

0< - Das FORTH-Wort **0<** überprüft, ob der oberste Stackeintrag kleiner als 0 ist. Es entfernt diesen obersten Eintrag vom Stack und ersetzt ihn durch ein Flag, das "wahr" ist, wenn die zuvor auf dem Stack befindliche Zahl kleiner Null ist; war sie gleich oder größer als Null, dann wird das Flag "falsch". Die Stack-Relation sieht folgendermaßen aus:

n --> n_{flag} (4-2)

Das Flag wird nur dann "wahr", wenn n kleiner Null ist. Beachten Sie die Ähnlichkeit in der Funktionsweise von **0=** und **0<**.

0> - Auch das FORTH-Wort **0>** entfernt die oberste Zahl vom Stack und legt statt dessen dort ein Flag ab. Dieses ist "wahr", wenn die Testzahl positiv war. Ist die Zahl kleiner oder gleich Null, dann wird das Flag "falsch". Die Stack-Relation sieht folgendermaßen aus:

n --> n_{flag} (4-3)

Das Flag wird nur "wahr", wenn n größer Null ist; in allen anderen Fällen ist es "falsch".

Als Beispiel wollen wir jetzt ein eigenes FORTH-Wort schreiben, das die oberste Zahl vom Stack entfernt und durch ein Flag ersetzt. Dieses soll "wahr" sein, wenn die Testzahl kleiner oder gleich Null war und in allen anderen Fällen "falsch". Man kann dies durch folgende Definition erreichen:

```
: KGN DUP 0< SWAP 0= + 0> ;                    (4-4)
```

Die Definition trägt den Namen **KGN** ("kleiner/gleich Null"). In ihr werden zwei Vergleichsbefehle eingesetzt. Als erstes duplizieren wir die fragliche Zahl. Dann wird das Wort **0<** aufgerufen. Dieses legt eine 1 auf den Stack, falls die Originalzahl kleiner Null war, andernfalls ist ihr Ergebnis Null. Das nächste Wort, **SWAP**, bringt die Originalzahl wieder an oberste Stack-Position. Daraufhin wird **0=** ausgeführt. Jetzt befindet sich eine 1 auf dem

Stack, wenn die Originalzahl gleich Null war, ansonsten steht dort eine Null. Die zweite Zahl auf dem Stack ist das Flag, welches das Wort **0<** dort hinterlassen hat. Wenn nun eines der beiden Stack-Elemente (oder auch beide) den Wert 1 hat, dann wollen wir ein "wahres" Flag auf den Stack pushen (nachdem wir zuvor die beiden Stack-Einträge entfernt haben). Dazu führen wir den Befehl + aus. Jetzt befindet sich statt der beiden obersten Stack-Einträge deren Summe auf dem Stack. Falls diese Zahl größer oder gleich 1 ist, dann soll das Ergebnis der gesamten Definition **KGN** "wahr" sein. Ist die Zahl jedoch gleich Null, dann soll auch das Ergebnis Null sein. Dafür sorgt das Wort **0>**. Es entfernt den obersten Stack-Eintrag und legt statt seiner ein Flag dort ab. Dieses hat den Wert "wahr", wenn die Ausgangszahl positiv war, und ist ansonsten "falsch". Wir haben also genau das erreicht, was wir wollen.

Im Folgenden betrachten wir einige FORTH-Wörter, die Entscheidungen in Abhängigkeit vom Wert zweier Zahlen treffen. Diese Wörter entfernen dazu stets die obersten beiden Einträge vom Stack. An ihrer Stelle legen sie dort ein Flag ab.

= - Das FORTH-WORT = entfernt die obersten zwei Stack-Einträge und ersetzt sie durch ein Flag. Das Flag ist "wahr", wenn die zwei vom Stack entfernten Zahlen gleich waren; ansonsten ist es "falsch". Wir haben also folgende Stack-Relation:

$$n_1 \; n_2 \; \text{-->} \; n_{flag} \qquad (4\text{-}5)$$

Das Flag wird "wahr", falls $n_1 = n_2$ und "falsch" in allen anderen Fällen. Ein Stack-Diagramm, das die Ausführung von = zeigt, sehen Sie in Abbildung 4-2.

< - Das FORTH-Wort < entfernt die beiden obersten Stack-Einträge und ersetzt sie durch ein Flag. Dieses ist "wahr", falls die zweite Zahl auf dem Stack kleiner als der oberste Stack-Eintrag war. Ansonsten - wenn die zweite Zahl größer oder gleich der ersten war - wird das Flag "falsch". Die Stack-Relation dazu:

$$n_1 \; n_2 \; \text{-->} \; n_{flag} \qquad (4\text{-}6)$$

```
I-----I   I-----I
I 15  I   I  1  I
I-----I   I-----I
I 15  I   I 15  I
I-----I   I-----I
I 27  I   I     I
I-----I   I-----I
I     I   I     I
I-----I   I-----I
I  .  I   I  .  I
I  .  I   I  .  I
I  .  I   I  .  I
I     I   I     I
I     I   I     I
I     I   I     I

   a)        b)
```

ABBILDUNG 4-2: a) Ein typischer Stack; b) der Stack nach Ausführung von =.

Das Flag wird also "wahr", wenn n_1 kleiner n_2 ist. Ansonsten ist das Flag "falsch".

> - Das FORTH-Wort > entfernt die beiden obersten Stack-Einträge und ersetzt sie durch ein Flag. Das Flag ist "wahr", wenn die zweite Zahl auf dem Stack größer als die erste ist; ansonsten ist es "falsch". Die Stack-Relation für > lautet:

$$n_1 \; n_2 \; \text{-->} \; n_{flag} \qquad (4\text{-}7)$$

Das Flag ist "wahr", wenn n_1 größer als n_2 ist. Ansonsten ist das Flag "falsch".

Wir wollen als Beispiel ein eigenes FORTH-Wort definieren, das zwei Zahlen vom Stack entfernt und an ihrer Stelle ein Flag dort ablegt. Dieses soll "wahr" sein, falls die zweite Zahl auf dem Stack kleiner oder gleich der ersten ist. In allen anderen Fällen - wenn also die zweite Zahl auf dem Stack größer als die erste war - soll das Flag "falsch" sein. Die Definition sieht folgendermaßen aus:

```
: KG 2DUP < ROT ROT = + 0> ;                    (4-8)
```

Gehen wir die Definition einfach einmal durch. Wir wollen insgesamt zwei logische Vergleiche durchführen. Dazu müssen wir zwei einfach genaue Integers duplizieren. Das können wir ebensogut mit **2DUP** machen. Als nächstes vergleichen wir diese zwei Zahlen über <. Dieses Kommando entfernt die beiden Zahlen vom Stack und ersetzt sie durch ein Flag. Das Flag ist "wahr", wenn der zweite Stack-Eintrag kleiner als der erste Stack-Eintrag war. Ansonsten ist es "falsch". Als nächstes führen wir zweimal den Befehl **ROT** aus. Jetzt befindet sich wieder das ursprüngliche Zahlenpaar an den obersten beiden Stack-Positionen. Der nächste Vergleich erfolgt mit Hilfe des Befehls =. Deshalb ist es egal, in welcher Reihenfolge die beiden Zahlen an die oberste Stack-Position gebracht werden. Wie Sie wissen, entfernt = die beiden obersten Stack-Einträge und ersetzt sie durch ein Flag. Dieses ist nur dann "wahr", wenn die beiden vom Stack entfernten Zahlen gleich sind. Jetzt haben wir zwei Flags in unserem Stack. Von da ab geht es weiter wie in der Definition (4-4). Die Befehle + und 0> werden ausgeführt. Somit befindet sich das gewünschte Ergebnis auf dem Stack.

Es gibt noch einige weitere Vergleichsbefehle, die jedoch noch nicht Teil von FORTH-79 sind. Sie finden sich aber im MMSFORTH und anderen FORTH-Systemen, weswegen wir sie hier besprechen.

Das FORTH-Wort <> funktioniert genau umgekehrt wie das bereits bekannte =. Es legt nämlich dann ein "wahres" Flag auf den Stack, wenn dessen beide oberste Einträge ungleich sind. Sind sie jedoch gleich, dann ist das Ergebnis von <> ein "falsches" Flag.

Das Wort <= funktioniert ähnlich wie <, außer daß das Flag nun "wahr" wird, wenn die zweite Zahl auf dem Stack kleiner oder <u>gleich</u> der Zahl an oberster Stack-Position ist.

Das FORTH-Wort >= funktioniert ähnlich wie >, außer daß das Flag nun "wahr" wird, wenn die zweite Zahl auf dem Stack größer oder <u>gleich</u> der Zahl an oberster Stack-Position ist.

4.2 Bedingte Verzweigungen

Wie werden jetzt erörtern, wie die Entscheidungsbefehle aus dem letzten Abschnitt dazu dienen können, in einem Programm Verzweigungen zu steuern. Dadurch wird unsere Programmierfähigkeit beträchtlich gesteigert.

IF und **THEN** - In FORTH gibt es zwei Kommandos, die für die Verzweigung in Programmen grundlegend sind. Sie lauten **IF** sowie **THEN**. Die beiden Wörter treten immer zusammen auf. Sie haben die allgemeine Form:

IF anweisungen a THEN anweisungen b (4-9)

Die beiden Wörter funktionieren folgendermaßen: Bei Ausführung von **IF** wird die oberste Zahl vom Stack entfernt und als Flag betrachtet. Falls (englisch "if") das Flag "wahr" ist, dann werden die "anweisungen a" ausgeführt, die hinter dem Wort **IF** folgen. Als nächstes werden die "anweisungen b" ausgeführt. War das zuvor auf dem Stack befindliche Flag jedoch falsch, dann werden die "anweisungen a" übergangen und es kommt nur zur Ausführung der "anweisungen b". Beachten Sie, daß das Flag (die Bedingung) vor dem **IF** kommen muß. Diese Regelung steht im Einklang mit der üblichen umgekehrten polnischen Notation, entspricht aber nicht dem umgangssprachlichen Gebrauch des "wenn...dann". Das Wort **IF** entfernt das Flag vom Stack; ist es "wahr", dann wird der Programmzweig zwischen den Wörtern **IF** und **THEN** begangen. Bei einem "falschen" Flag werden diese Befehle übergangen, und nur der Programmteil hinter **THEN** gelangt zur Ausführung. Beachten Sie, daß die auf **THEN** folgenden Wörter also in jedem Fall ausgeführt werden.

Wir wollen einmal ein Beispiel für ein FORTH-Wort analysieren, das zwei Programmzweige enthält. Das Wort vergleicht zwei Zahlen auf dem Stack. Sind sie gleich, dann gibt es die Meldung aus: "Die Zahlen sind gleich". Bei Ungleichheit der Zahlen meldet das Programm jedoch: "Die Differenz lautet" und gibt daraufhin die Differenz der beiden Zahlen aus. Wir erreichen dies durch die Definition in Abbildung 4-3.

```
 0 ( Eine einfache Programmverzweigung )
 1 : CHECKEQ  2DUP  =
 2            IF  ." Die Zahlen sind gleich "  2DROP  QUIT
 3            THEN   -      ." Die Differenz lautet " . ;
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

ABBILDUNG 4-3: Ein einfaches Beispiel für Programmverzweigungen

Zeile 1 dieses Programms definiert den Namen des neuen Wortes, nämlich **CHECKEQ**. Da unter Umständen zwei Operationen mit den fraglichen Integers erforderlich sind, duplizieren wir sie beide mittels **2DUP**. Als nächstes wird das oberste Zahlenpaar vom Stack entfernt und mittels = verglichen. Sind die Zahlen gleich, dann wird ein Flag mit dem Wert "wahr" auf den Stack gepusht. Ansonsten ist das Flag "falsch". Jetzt kommt das Wort **IF** an die Reihe. Es entfernt das Flag an oberster Stack-Position. Im Falle eines "wahren" Flags werden dann die Wörter zwischen **IF** und **THEN** ausgeführt. Dies führt dazu, daß die Meldung "Die Zahlen sind gleich" ausgegeben wird. Da im Folgenden der Stack nicht mehr benötigt wird, führen wir das Kommando **2DROP** aus, um ihn zu bereinigen. Anschließend kommt der Befehl **QUIT** an die Reihe, wodurch das Wort verlassen und die Kontrolle wieder an Ihr Terminal übergeben wird. Wie Sie sehen, kann also das Wort **QUIT** durchaus Bestandteil eines Programmzweigs sein. Wenn in unserem selbstdefinierten Wort der Programmzweig mit dem **QUIT** ausgeführt wird, dann gelangen die Programmschritte hinter **THEN** nicht zur Ausführung. Beachten Sie aber, daß ein Aufruf von **CHECKEQ** aus einem anderen FORTH-Wort heraus in diesem Fall dazu führt, daß alle Berechnungen durch **QUIT** abgebrochen werden und nicht nur das Wort **CHECKEQ** verlassen wird. Sie müssen also beim Umgang mit **QUIT** sehr vorsichtig sein.

Jetzt wollen wir einmal annehmen, daß das Flag "falsch" war. In diesem Fall werden die Befehle zwischen **IF** und **THEN** übergangen. Nun berechnet das FORTH-Wort - die Differenz der beiden Zahlen. Diese ersetzt die beiden Originalzahlen auf dem Stack. Als letztes führen wir den Befehl ." aus und sorgen dafür, daß die Meldung "Die Differenz lautet" auf dem Bildschirm ausgegeben wird. Der Punktbefehl bringt dann die Differenz zum Vorschein. Da sowohl das Wort - als auch . jeweils eine Zahl vom Stack entfernen, brauchen wir in diesem Programmzweig nicht mittels **DROP** den Stack zu bereinigen.

ELSE - Programmverzweigungen werden durch das FORTH-Wort zu einem wesentlich wirkungsvolleren Instrument. Der Befehl **ELSE** kann folgendermaßen in die Befehlsfolge **IF...THEN** eingebaut werden:

IF anweisungen a ELSE anweisungen b THEN anweisungen c (4-10)

In diesem Fall werden folgende Programmschritte unternommen. Wieder gehen wir davon aus, daß der oberste Stack-Eintrag ein Flag ist. Ist das Flag "wahr", dann werden die "anweisungen a" ausgeführt, nicht aber die "anweisungen b". Als nächstes kommen dann die "anweisungen c" an die Reihe. Ist das Flag aber "falsch", dann werden die "anweisungen a" übergangen, es gelangen jedoch die "anweisungen b" sowie die "anweisungen c" zur Ausführung. Nocheinmal: Wenn das Flag "wahr" ist, dann werden die Wörter zwischen **IF** und **ELSE** ausgeführt, während die zwischen **ELSE** und **THEN** übergangen werden. Ist das Flag "falsch", dann werden die Anweisungen zwischen **IF** und **ELSE** übergangen, während die zwischen **ELSE** und **THEN** stehenden Wörter ausgeführt werden. Das Wort **IF** besitzt natürlich auch eine Stack-Relation; sie lautet:

n_{flag} --> (4-11)

Wir wollen die Arbeitsweise des Wortes **ELSE** anhand des Programms in Abbildung 4-4 verdeutlichen. Dieses einfache Programm *könnte* z.B. ein Lehrer dazu benutzen, Informationen über die Leistungen seiner Schüler anhand der erreichten Punktzahl auszudrucken. Die Punktzahl ist eine Integer, die sich an oberster Stack-Position befinden soll. Bei Ausführung des Wortes **NOTEN** passiert dann folgendes. Bei einer Punktzahl kleiner 60 wird die Meldung "Der

Schüler fällt durch" ausgegeben. Ist die Punktzahl größer oder gleich 60, dann erscheint statt dessen "Der Schüler besteht". In beiden Fällen wird jedoch die Differenz zwischen der erreichten Punktzahl und der maximalen Punktzahl von Hundert zusammen mit einer erklärenden Meldung ausgegeben.

```
 0 ( Einfache Bewertung durch Verzweigung )
 1 : NOTEN      ." Der Schueler " 60 -       0<
 2              IF ." faellt durch "  ELSE ." besteht "
 3              THEN 40 - NEGATE  ." erst mit " .
 4              ." weiteren Punkten bist du perfekt! "   ;
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

ABBILDUNG 4-4: Ein einfaches Beispiel für bedingte Verzweigungen

Sehen wir uns das Wort **NOTEN** in allen Einzelheiten an (vgl. Abb. 4-4). In Zeile 1 wird zuerst einmal der Text "Der Schüler" ausgegeben. Daraufhin pushen wir die Zahl 60 auf den Stack. Das Wort - sorgt nun dafür, daß die Punktzahl des Schülers, vermindert um den Wert 60, als neues oberstes Stack-Element gegeben ist. Ist diese Zahl negativ, dann ist der Schüler durchgefallen. Da wir diese Zahl zweimal brauchen, duplizieren wir sie jedoch zuerst mit **DUP**. Dann wird der Vergleich mit **0<** ausgeführt. Die oberste Zahl auf dem Stack, also die Punktzahl des Schülers minus 60, wird von diesem entfernt und durch ein Flag ersetzt. Dieses hat den Wert "wahr", wenn die Ausgangszahl kleiner Null ist oder, anders gesagt, wenn der Schüler weniger als 60 Punkte erreicht hatte. Als nächstes stoßen wir auf den Befehl **IF**. Dieser entfernt das Flag vom Stack. Ist es "falsch", dann wird die Meldung "fällt durch" ausgegeben. Außerdem werden die Anweisungen zwischen **ELSE** und **THEN** übergangen. Wenn andererseits das Flag "falsch" ist, dann ignoriert FORTH die Wörter zwischen **IF** und **ELSE** und gibt

somit die Meldung "besteht" aus. Jetzt haben wir als obersten Eintrag auf dem Stack wieder die Ausgangszahl (Punktzahl des Schülers minus 60). Das kommt von der **DUP**-Anweisung in Zeile 1. Jetzt legen wir eine 40 auf den Stack und führen den Befehl - aus. Dieser ersetzt in bereits bekannter Weise die beiden Zahlen durch ihre Differenz. Das neue oberste Stack-Element ist somit die Punktzahl des Schülers minus 100. Da es sich dabei um eine negative Zahl (oder die Null) handelt, drehen wir deren Vorzeichen mittels **NEGATE** um. Damit wird die Zahl auf dem Stack positiv. Dies ist die Zahl, die man auf die ursprüngliche Punktzahl des Schülers addieren muß, um 100 zu erhalten. Angenommen, es handelt sich dabei um eine 15. In diesem Fall sorgt das restliche Programm dafür, daß die Meldung "Erst mit 15 weiteren Punkten bist du perfekt" ausgegeben wird.

4.2.1 Verschachtelte Kontrollstrukturen

Wie wir gesehen haben, lassen sich in einem Programm über die Befehlsfolge **IF-ELSE-THEN** zwei Zweige einführen, einer zwischen **IF** und **ELSE**, der andere zwischen **ELSE** und **THEN**. Innerhalb eines solchen Programmzweigs kann nun eine weitere Verzweigung stattfinden. Diese Art Programmstruktur bezeichnet man mit dem Fachausdruck verschachtelte Kontrollstruktur. Den Einsatz verschachtelter Kontrollstrukturen wollen wir an einem FORTH-Wort illustrieren, das der Punktzahl von Schülern die entsprechende Bewertung zuordnet. Das Programm erwartet, daß sich die Punktzahl als oberster Eintrag auf dem Stack befindet. Dann wird das neue Wort **BENOTUNG** ausgeführt. Bei einer Punktzahl von unter 60 gibt dieses Wort die Meldung aus: "NOTE: DURCHGEFALLEN". Bei einer Leistung von mehr als 60 und weniger als 70 Punkten gibt es die Meldung aus: "NOTE: VIER"; zwischen 70 und 80 Punkten wird ausgegeben: "NOTE: DREI". Bei einer Punktzahl zwischen 80 und 90 erscheinen die Wörter "NOTE: ZWEI" und zwischen 90 und 100: "NOTE: EINS".

Das zugehörige Programm sehen Sie in Abbildung 4-5. Das neu definierte Wort trägt den Namen **BENOTUNG**. Zeile 1 gibt zuerst einmal den Text "NOTE:" aus. Als nächstes wird eine 60 auf dem Stack abgelegt und mittels - von der Punktzahl des Schülers subtrahiert. Der oberste Stack-Eintrag stellt nunmehr also die Punktzahl des Schülers weniger 60 dar. Diese Zahl duplizieren wir mittels **DUP**. Als nächstes stellen wir den Vergleich **0<** an.

```
 0 ( Verschachtelte Kontrollstrukturen )
 1 : BENOTUNG    ." NOTE: " 60 -     DUP     0<
 2       IF   ." DURCHGEFALLEN "  DROP
 3          ELSE    10 - DUP   0<    IF
 4             ." VIER "  DROP
 5             ELSE   10 -  DUP   0<   IF   ." DREI " DROP
 6                ELSE     10 -    0<   IF ." ZWEI "
 7                   ELSE    ." EINS "     THEN
 8              THEN
 9          THEN
10      THEN    ;
11
12
13
14
15
```

ABBILDUNG 4-5: Ein FORTH-Wort mit verschachtelten Kontrollstrukturen

Wenn das Flag, das dieser Vergleich als Ergebnis auf dem Stack hinterläßt, "wahr" ist, dann bedeutet dies, daß die Punktzahl des Schülers schlechter als 60 sein muß. In diesem Fall geben wir die Meldung "DURCHGEFALLEN" aus. Es werden nun alle Anweisungen zwischen **ELSE** und **THEN** ignoriert. Bei diesem Punkt müssen wir besonders vorsichtig sein. Der Zweig zwischen **ELSE** und **THEN** enthält nämlich ein weiteres Vorkommnis von **IF**. Wir haben also den Fall, daß innerhalb einer Programmverzweigung eine weitere Programmverzweigung eingebettet ist. Jedes **IF** innerhalb einer solchen geschachtelten Konstruktion bezieht sich auf eine ganz bestimmte **ELSE**-und-**THEN**-Folge. Wir müssen die zusammengehörigen **IF**-, **ELSE**- und **THEN**-Befehle sorgfältig zusammen gruppieren. Dazu kann man sich des folgenden Verfahrens bedienen.

Als erstes machen wir das <u>innerste</u> (am weitesten eingeschachtelte) **IF** ausfindig. Die auf dieses folgende <u>nächste</u> Kombination von **ELSE** und **THEN** bezieht sich nun auf dieses innerste **IF**. Vom innersten **IF** gehen wir nach außen und finden das vorletzte **IF**. Die <u>nächste</u> Kombination von **ELSE** und **THEN** hinter der soeben gefundenen bezieht sich nun auf dieses **IF**. Auf diese Art arbeiten wir von innen nach außen und stellen so die Beziehung zwischen den einzelnen **IF**, **ELSE** und **THEN** her. Sie können auch noch ein anderes Verfahren anwenden. Dazu gruppieren wir das erste **IF** mit dem

nächsten **ELSE** und dem letzten **THEN** in der verschachtelten Struktur (Beachten Sie, daß dies nicht notwendigerweise das letzte **THEN** im gesamten Wort sein muß!). Dann wiederholen wir dieses Vorgehen mit den verbleibenden **IF**, **ELSE** und **THEN**. So gehört z.B. in Abbildung 4-5 das **IF** auf Zeile 2 zum **ELSE** auf Zeile 3 und dem **THEN** auf Zeile 10. Das erste **IF** im Wort wird also auf das nächste **ELSE** und das letzte **THEN** bezogen.

Wenn also in unserem Programmbeispiel das Original-Flag den Wert "wahr" hat, dann wird "DURCHGEFALLEN" ausgegeben, anschließend mittels **DROP** der Stack bereinigt und das Wort beendet (denn hinter dem zugehörigen **THEN** stehen keine weiteren Befehle mehr). Ist andererseits das Flag "falsch", dann werden die Befehle zwischen dem **IF** in Zeile 2 und dem **ELSE** in Zeile 3 ignoriert. Statt dessen gelangen die Wörter zwischen dem **ELSE** von Zeile 3 und dem **THEN** von Zeile 10 zur Ausführung. Die Zahl, die sich zu diesem Zeitpunkt an oberster Stack-Position befindet, ist die duplizierte Punktzahl des Schülers weniger 60. Jetzt legen wir in Zeile 3 eine 10 auf den Stack und subtrahieren erneut. Die Zahl, die wir daraus als Ergebnis erhalten, ist nur dann größer oder gleich Null, wenn die Ausgangspunktzahl größer oder gleich 70 war. Diese neue Zahl duplizieren wir ebenfalls in Zeile 3 mittels **DUP**. Dann kommt das Vergleichswort **0<** an die Reihe und legt ein Flag auf den Stack. Dieses wird vom nächsten **IF** entfernt. Hatte es den Wahrheitswert "wahr", dann wird eine Vier ausgegeben.

Versuchen Sie einmal, das gerade aktuelle **IF** mit seinem dazugehörigen **ELSE** und **THEN** in Beziehung zu bringen. Wie wir wissen, gehört zu dem **IF** von Zeile 3 das **ELSE** auf Zeile 5 und das **THEN** in Zeile 9. Bei einem "wahren" Flag wird also eine "VIER" ausgegeben, anschließend mittels **DROP** der Stack bereinigt und die Programmkontrolle an Zeile 9 und anschließend an Zeile 10 übergeben, was bedeutet, daß das Wort endet. Ist das Flag jedoch "falsch", dann werden die Befehle zwischen dem **IF** auf Zeile 3 und dem **ELSE** von Zeile 5 ignoriert. Statt dessen gelangen die Wörter zwischen dem **ELSE** in Zeile 5 und dem **THEN** in Zeile 9 zur Ausführung. Hier verfolgen wir wieder dieselben Schritte wie bereits zuvor und geben "DREI" oder "ZWEI" aus, je nachdem, welche Punktzahl der Schüler erreicht hat. Beachten Sie, daß bei Ausführung des **IF** von Zeile 6 der Schüler entweder eine 2 oder eine 1 erhält. Wenn das Flag, das sich durch das Ergebnis des **0<** in Zeile 6 ergibt, "wahr" ist, dann sollte eine 2 ausgegeben werden, anderenfalls wird eine 1 auf dem Bildschirm erscheinen.

FORTH bedient sich des Returnstacks, um die Abarbeitung solcher verschachtelter Kontrollstrukturen wie in Abbildung 4-5 zu bewältigen. Kommen bei verschachtelten Verzweigungen die Wörter **>R** und **R>** vor, dann müssen Sie äußerste Sorgfalt walten lassen. In jedem Zweig einer solchen Struktur muß eine gleiche Anzahl von **>R** und **R>**-Worten zur Ausführung gelangen. Beachten Sie, daß es dazu nicht reicht, lediglich eine gleiche Anzahl von **>R** und **R>**-Worten im Programm stehen zu haben. Es müssen auch gleich viele dieser Wörter ausgeführt werden. Dies gilt für jeden möglichen Zweig in Ihrem Programm. Selbst wenn nur zwei Zweige existieren, also keine Verschachtelungen vorgenommen werden, müssen immer noch gleichviel **R>** und **>R**-Befehle ausgeführt werden. Wenn Sie sich nicht an diese Vorschrift halten, dann können Sie Ihr System zum Absturz bringen. Sie müßten es dann erneut starten und würden unter Umständen alle Daten verlieren, wenn Sie diese nicht auf Diskette gesichert haben.

4.3 Unbedingte Schleifen

Oft ist es nötig, daß Ihr Programm eine Folge von Schritten wiederholt ausführt. Man spricht in diesem Fall von einer Programmschleife. Wir könnten ein FORTH-Wort schreiben, das aus einer Folge von Befehlen besteht und dann ein anderes FORTH-Wort, das dieses erste FORTH-Wort wiederholt aufruft. Dies ist aber höchstens eine Notlösung, besonders, wenn die Operationen sehr viele Male wiederholt werden müssen. In FORTH gibt es für diesen Zweck spezielle Wörter, mit denen Programmschleifen ausgelöst werden können, ohne daß der Programmierer die Wiederholung der Befehle von Hand durch Hinschreiben bewirken muß. Das Prinzip der Programmschleifen eröffnet noch einige weitere nützliche Möglichkeiten, die wir in diesem Abschnitt besprechen wollen.

DO und **LOOP** - Diese beiden Wörter werden eingesetzt, um elementare Schleifen in Programmen zu bewerkstelligen. Eine solche Schleife hat folgenden allgemeinen Aufbau

n_1 n_2 DO anweisungen a LOOP (4-12)

Dieses Konstrukt hat die folgende Bedeutung: Bei Ausführung einer Anweisung der Form (4-12) werden die Werte n_1 und n_2 vom Stack entfernt und auf dem Return-Stack abgelegt, wobei dort n_2 zuoberst zu liegen kommt. (Was die Werte n_1 und n_2 betrifft, so müssen Sie sich um den Return-Stack nicht kümmern. Nach Verlassen der Schleife wird dieser nämlich automatisch von n_1 und n_2 bereinigt.) Als nächstes werden die "anweisungen a" ausgeführt. Trifft der FORTH-Interpreter nun auf das Wort **LOOP**, so erhöht er den Wert von n_2 auf dem Return-Stack um 1 und vergleicht diesen neuen Wert mit n_1. Falls n_2 kleiner als n_1 ist, werden erneut die "anweisungen a" wiederholt. Jedesmal beim Erreichen des Wortes **LOOP** wird also n_2 um 1 erhöht (inkrementiert) und getestet. Wird es dabei einmal gleich oder größer n_1, dann wird die Schleife verlassen, und die Anweisungen hinter **LOOP** werden ausgeführt. Ist andererseits n_2 immer noch kleiner als n_1, dann werden die "anweisungen a" erneut ausgeführt. Deswegen sorgen die folgenden Befehle

```
7 4 DO ." HALLO " LOOP                                    (4-13)
```

dafür, daß das Wort HALLO dreimal auf dem Bildschirm ausgegeben wird.

Die Stack-Relation bei Ausführung eines **DO** lautet:

$$n_1\ n_2 \; \text{-->} \qquad (4\text{-}14)$$

Die Zahl n_1 heißt <u>Testwert</u> der Schleife; sie bleibt unverändert. Die Zahl n_2 bezeichnet man als <u>Schleifenindex</u>. Sie wird bei jedem Durchgang durch die Schleife um 1 erhöht. Beachten Sie, daß mindestens ein Durchgang durch die Schleife vorgenommen wird, da der Schleifenindex erst bei Erreichen von **LOOP** mit dem Testwert verglichen wird.

Man kann sich den Schleifenindex vom Return-Stack auf den Parameter-Stack mit Hilfe des FORTH-Wortes **I** kopieren. Auch den Testwert kann man vom Return-Stack auf den Parameter-Stack bringen, und zwar mit dem Wort **I'**. Beachten Sie, daß keine dieser beiden Operationen etwas am Return-Stack verändert. Wir haben sie bereits in Abschnitt 2-6 kennengelernt. Als Beispiel für ein Pro-

gramm mit Schleife wollen wir ein FORTH-Wort schreiben, das die Fakultät einer Integer an oberster Stack-Position berechnet. In der mathematischen Fachliteratur stellt man die Fakultät einer Zahl mit Hilfe des Ausrufezeichens dar. Als Beispiel

5!=5*4*3*2*1=120.

Entsprechend ist 3!=3*2*1=6. (Beachten Sie, daß man "5!" als "5-Fakultät" liest). Die Definition des FORTH-Wortes **FACT** entnehmen Sie bitte der Abbildung 4-6. Wir wollen uns einmal ansehen, wie dieses Wort funktioniert. Zur Berechnung von 5! geben wir ein:

5 FACT (RETURN) (4-15)

Das bedeutet, daß sich bei Aufruf von **FACT** eine 5 auf dem Stack befindet. Die Definition von **FACT** enthält alle notwendigen Anweisungen, um das Schleifenkonstrukt "in Gang zu setzen". Dabei soll bei jedem Durchgang durch die Schleife eine Multiplikation stattfinden. Der Anfangswert des Schleifenindex ist 1. Wenn wir die Fakultät von 5 berechnen wollen, dann sollten wir die Schleife fünfmal durchlaufen. Deshalb benötigen wir einen Testwert von 6. Dies kommt daher, daß die Schleife erst dann verlassen wird, wenn der erhöhte Index dem Testwert entspricht.

Jetzt zur Abbildung 4-6. Als erstes pushen wir 1 auf den Stack und rufen das Wort +. Dadurch wird der Ausgangswert um 1 erhöht. Wir hätten dies natürlich genauso mit Hilfe des Befehls **1+** erreichen können. Dann legen wir eine weitere 1 auf den Stack. Zu diesem Zeitpunkt der Programmausführung haben wir also zuoberst auf dem Stack eine 1, gefolgt von der Ausgangszahl, welche um 1 erhöht wurde - in diesem Fall also die 6. Nun legen wir eine weitere 1 auf den Stack. Diese brauchen wir in unseren Berechnungen. Ehe wir damit aber beginnen können, müssen wir den Anfangswert des Schleifenindex sowie den Testwert an die richtigen Stack-Positionen, also an erste und zweite Position bringen. Dies geschieht durch zweimalige Ausführung von **ROT**. In unserem Beispiel 4-15 bedeutet dies, daß nach Ausführung der zwei **ROT**-Wörter der Stack folgendes Aussehen hat: 1 6 1. Dabei ist die letzte Zahl der oberste Stack-Eintrag. Jetzt kommt endlich das **DO** an die Reihe, welches die Schleife einleitet. **DO** entfernt die obersten beiden Stack-Werte und legt sie auf den Return-Stack. Infolgedessen befindet sich auf dem Daten-Stack jetzt zuoberst eine 1.

```
0     ( Berechnung der Fakultaet )
1     : FACT   1  +  1  1  ROT  ROT
2              DO  I  *  LOOP .  ;
3
4
5
6
7
8
9
10
11
12
13
14
15
```

ABBILDUNG 4-6: Ein Fakultätsprogramm

Betrachten wir nun die Anweisungen zwischen **DO** und **LOOP**. Die erste ist der Befehl I. Diese dupliziert den Schleifenindex an die oberste Daten-Stack-Position. Anschließend wird mittels * multipliziert. Somit werden die obersten beiden Stack-Einträge durch ihr Produkt ersetzt. Beim ersten Schleifendurchgang berechnen wir also 1*1. Beim zweiten Durchgang durch die Schleife hat der Index den Wert 2. Deshalb wird das Produkt 1*2=2 berechnet. Der dritte Schleifendurchgang arbeitet mit einem Schleifenindex von 3 und berechnet 2*3=6. Nach Abschluß der Schleife haben wir also die gewünschte Fakultät berechnet. Dieser Wert wird nach Beendigung der Schleife durch das Punktkommando ausgegeben.

4.4 Schleifen-Inkrement mittels +LOOP

Wir haben gesehen, daß bei jedem Durchgang durch eine Schleife der Index automatisch um 1 erhöht wird. Manchmal kann es aber wünschenswert sein, den Schleifenindex um einen anderen Betrag als 1 zu verändern. Dazu ersetzen wir einfach das FORTH-Wort **LOOP** durch ein anderes Wort, welches dies bewerkstelligt. Dieses Wort lautet **+LOOP**. Bei Ausführung von **LOOP** wird der Schleifenindex um 1 erhöht, und anschließend wird überprüft, ob sein Betrag bereits

den Testwert übersteigt. Anders bei **+LOOP**: Bei Ausführung dieses Wortes wird der oberste Stack-Eintrag vom Stack entfernt und dient jetzt als Schleifeninkrement, d.h., er gibt den Betrag an, um den der Schleifenindex erhöht werden soll. Nachdem der Schleifenindex um das Schleifeninkrement erhöht wurde, wird der neue Indexwert mit dem Testwert verglichen, wobei FORTH nachsieht, ob der Schleifenindex kleiner als der Testwert ist. (Dies gilt nur für den Fall, daß das Schleifeninkrement positiv war.) Die Stack-Relation für das Wort **+LOOP** lautet

```
n -->                                              (4-16)
```

Als Beispiel für dieses Wort werden wir ein kleines Programm schreiben, welches die Summe aller ungeraden Zahlen angibt, die innerhalb eines durch die zwei obersten Stack-Einträge festgelegten Bereiches liegen. Wir geben diesem neuen Wort den Namen **ODDSUM.** Beispielsweise sollte

```
17 3 ODDSUM (RETURN)                               (4-17)
```

die Summe der ungeraden Zahlen zwischen 3 und 17 einschließlich auf dem Bildschirm ausgeben. Die Definition von **ODDSUM** lautet:

```
: ODDSUM SWAP 1+ 0 SWAP ROT DO I + 2 +LOOP     (4-18)
```

Die erste Operation in diesem Wort ist ein Austausch der beiden obersten Stack-Werte mittels **SWAP.** Im Beispiel (4-17) bringt dies die Zahl 17 an oberste Stack-Position. Sie wird anschließend mittels **1+** um eins erhöht. Dann bringen wir noch eine 0 auf den Stack und führen erneut **SWAP** aus. Das anschließende Wort **ROT** führt dann dazu, daß wir für Beispiel (4-18) den Stack 0 18 3 erhalten. Bei dem Einstieg in die **DO**-Schleife haben wir also (in Beispiel (4-17)) einen Anfangswert für den Schleifenindex von 3, während der Testwert 18 ist. **DO** entfernt diese beiden Zahlen vom Stack, so daß sich nun die Null dort zuoberst befindet. Bei einem Durchgang durch die Schleife legt I den Schleifenindex auf den Stack. Dieser ist in unserem Beispiel beim ersten Durchgang gleich 3, so daß innerhalb der Schleife auf die 0 eine 3 addiert wird. Diese beiden Werte - 3 und 0 - werden entfernt und durch

ihre Summe ersetzt. Anschließend legen wir eine 2 auf den Stack. Sie sorgt zusammen mit dem Wort **+LOOP** dafür, daß der Schleifenindex um 2 erhöht wird. Deshalb wird beim nächsten Durchgang durch die Schleife eine 5 auf die bisher aufgelaufene Summe addiert. Wenn schließlich der erhöhte Schleifenindex größer als der Testwert der Schleife ist, dann endet die Ausführung der Programmschleife. Das Punktkommando gibt die gewünschte Summe aus.

Die bisherigen Beispielprogramme benutzten positive Werte für den Schleifenindex, den Testwert und das Schleifeninkrement. Dies muß jedoch nicht so sein, alle drei Parameter können genausogut negative Werte besitzen. Bei einem positiven Schleifeninkrement wird die Schleife verlassen, wenn der Schleifenindex größer oder gleich dem Testwert ist. Ist andererseits das Schleifeninkrement negativ, dann endet die Schleife, wenn der Schleifenindex einen Wert besitzt, der kleiner als der Testwert ist. Betrachten Sie z.B. das Wort **NEGTEST**:

```
: NEGTEST -5 2 DO I . -2 +LOOP ;                    (4-19)
```

Wenn wir dieses Wort aufrufen, so erhalten wir als Ergebnis

```
2 0 -2 -4 ok
```

LEAVE - Mit dem FORTH-Wort **LEAVE** kann man das Verlassen einer Schleife erzwingen. Bei Ausführung von **LEAVE** wird der Testwert der Schleife auf den aktuellen Wert des Schleifenindex gesetzt. Dadurch wird die Schleife bei dem nächsten **LOOP** verlassen. **LEAVE** wird fast immer in Zusammenhang mit Verzweigungen eingesetzt. Wir werden noch später in diesem Kapitel ein Anwendungsbeispiel für **LEAVE** kennenlernen.

4.5 Verschachtelte Schleifen

Eine Schleife kann innerhalb einer anderen eingeschachtelt sein. Wir wollen dies an einem kleinen Beispiel demonstrieren:

```
: NEST 10 1 DO 5 3 DO I . LOOP LOOP ;                (4-20)
```

Wenn wir das Wort **NEST** aufrufen, dann erhalten wir neunmal hintereinander die Zahlen 3 und 4 auf dem Bildschirm. Die Wortdefinition enthält zwei Schleifen - eine äußere Schleife, deren Index mit 1 beginnt und die einen Testwert von 10 hat, sowie eine innere Schleife, deren Index den Anfangswert 3 hat und die mit einem Testwert von 5 arbeitet. Bei Ausführung von **NEST** gehen wir zuerst einmal in die äußere Schleife. Dabei werden die Zahlen 5 und 3 auf den Stack gelegt. Als nächstes Wort leitet **DO** die innere Schleife ein. Dies bedeutet, daß die 5 und die 3 wieder vom Stack entfernt und auf den Return-Stack gepusht werden. Infolgedessen werden die 10 und die 1, die vor der äußeren **DO**-Schleife auf dem Stack waren, um zwei Positionen auf dem Returnstack nach unten wandern. Nun stoßen wir auf das FORTH-Kommando **I**. Dieses legt den Wert des Schleifenindex auf den Stack; beachten Sie dabei, daß **I** stets den Index der Schleife ausgibt, in der wir uns gerade befinden. In unserem Beispiel ist dies gerade die innerste Schleife. Beim ersten Durchgang durch die innerste Schleife hat der Index den Wert 3, weswegen auch dieser Wert auf den Stack gelegt wird. Das Punktkommando entfernt ihn wieder von dort und gibt ihn aus. Der nächste Schleifendurchgang erfolgt mit einem Schleifenindex von 4; dieser Wert wird gedruckt. Die innere Schleife bricht nun ab, und ihre Parameter werden vom Return-Stack entfernt. Wir befinden uns aber immer noch in einer Schleife, nämlich in der äußeren Schleife, deren Schleifenindex und Testwert jetzt zuoberst auf dem Return-Stack sind. Diese äußere Schleife durchlaufen wir nun zum zweitenmal. Wieder werden die 5 und die 3 auf den Stack gelegt und die innere **DO**-Schleife durchlaufen. Es wiederholen sich dieselben Vorgänge wie zuvor: 3 und 4 werden wieder ausgegeben, und die innere Schleife wird verlassen. Dieser ganze Vorgang wird so lange wiederholt, bis die äußere Schleife insgesamt neunmal durchlaufen wurde.

Wenden wir uns noch einmal dem I-Befehl zu. Er besorgt den Index der Schleife, die gerade ausgeführt wird, und legt ihn auf den

Daten-Stack. Ist die aktuelle Schleife die innerste, dann wird I deren Index auf den Stack legen. Ist die Ausführung der innersten Schleife jedoch abgeschlossen, dann dupliziert **I** den äußeren Schleifenindex auf den Stack. Ähnlich sorgt das FORTH-Wort **I'** dafür, daß der Testwert der gerade in Aktion befindlichen Schleife auf den Stack gelegt wird. Manchmal kommt es vor, daß wir uns in einer inneren Schleife befinden, aber den Index der äußeren Schleife auf den Stack gelegt haben wollen. Dies erreichen wir mit dem FORTH-Wort **J** (vergleiche Abschnitt 2-6). Beachten Sie, daß **I**, **I'** und **J** im Return-Stack nichts verändern. Nach Abschluß einer Schleife werden alle ihre Schleifenparameter vom Return-Stack entfernt. Sie brauchen sich darum nicht zu kümmern.

Bei eingeschachtelten Schleifen muß die innere Schleife vollständig innerhalb der äußeren Schleife liegen. Beachten Sie, daß jeder **DO**-Befehl mit einem zugehörigen **LOOP** oder **+LOOP** verbunden ist. Um herauszufinden, welches **DO** zu welchem **LOOP** bzw. **+LOOP** gehört, können Sie im wesentlichen genauso vorgehen, wie wir es bei **IF** und **THEN** kennengelernt haben. Machen Sie zuerst das innerste **DO** ausfindig. Dazu gehört das nächste vorkommende **LOOP** oder **+LOOP**. Dann suchen Sie das vorhergehende **DO**. Dazu gehört das nächste **LOOP**, ausgehend von dem zuletzt gefundenen **LOOP**. Wenn wir in dieser Weise vorgehen, so können wir den Geltungsbereich jeder Schleife ausfindig machen.

Als Beispiel für verschachtelte Schleifen wollen wir ein Programm schreiben, das pythagoreische Zahlentripel berechnet. Damit bezeichnen wir drei ganze Zahlen, die die Länge der drei Seiten eines rechtwinkeligen Dreiecks wiedergeben sollen. Die Zahlen a, b und c (allesamt Integers) stellen ein pythagoreisches Zahlentripel dar.

$$a^2 + b^2 = c^2 \tag{4-21}$$

Wenn etwa a = 3 und b = 4, dann ergibt sich c = 5, und somit ist 3, 4, 5 ein solches pythagoreisches Tripel. Nun gibt es nur eine Handvoll ganzer Zahlen, die die Gleichung (4-21) erfüllen. Wir wollen uns deshalb ein FORTH-Programm schreiben, das solche Tripel für Werte von a und b zwischen 1 und 100 berechnet. Das zugehörige Programm sehen Sie in Abbildung 4-7. Wir definieren uns zuerst zwei neue Wörter, die wir benötigen. Das erste, **SQUARE**, ist uns bereits aus Abschnitt 2-3 unter dem Namen **ZWEITE** bekannt.

Es entfernt die oberste Zahl von Stack und ersetzt sie durch ihr Quadrat (die zweite Potenz). Das zweite Wort, **PT**, druckt eine Zahl in einem Datenfeld mit einer Breite von 15 Spalten.

Die Definition des Wortes zur Berechnung des pythagoreischen Zahlentripels beginnt auf Zeile 3. Wir haben dem Wort den Namen **PYTHTRIP** gegeben. Es zeichnet sich durch drei verschachtelte Schleifen aus: eine äußere Schleife, in die zwei weitere Schleifen eingeschachtelt sind, nämlich eine mittlere Schleife (innerhalb der äußeren Schleife) und eine innerste Schleife, die in die anderen beiden eingebettet ist.

Wir wollen einmal sehen, wie unser Wort funktioniert. Die Zeile 3 legt erst einmal den Testwert 100 und den Schleifenindex 1 auf den Stack. Diese werden von **DO** entfernt und auf den Return-Stack gelegt. Damit sind alle nötigen Vorkehrungen für die äußere Schleife getroffen. Das nächste Wort, **I**, dupliziert den Laufindex der äußeren Schleife auf den Stack. Diesen verdoppeln wir durch **DUP** und lassen uns von **SQUARE** deren Quadrat berechnen. Nach Beendigung der Zeile 3 befinden sich also auf dem Stack der Index der äußeren Schleife sowie dessen Quadrat (das Quadrat ist zuoberst). Die nächste Zeile leitet die mittlere **DO**-Schleife ein. Es ist sehr wichtig, daß nach einem Durchgang durch die mittlere Schleife der oberste Stack-Eintrag unverändert bleibt. Er muß nach Vollendung der mittleren Schleife nach wie vor den Laufindex der äußeren Schleife und dessen Quadrat enthalten. Zeile 4 legt erst einmal die 100 auf den Stack. Anschließend besorgen wir uns mit **I** den Index der äußeren Schleife. Wir tun dies, um die mehrmalige Berechnung desselben Zahlentripels zu vermeiden. Wenn wir z.B. die Zahlen 3, 4 und 5 gefunden haben, dann wollen wir nicht auch noch 4, 3 und 5 finden. Dadurch wird unser Programm um einiges schneller. Nachdem so der Testwert und der Schleifenindex für die mittlere Schleife vorbereitet worden sind, holt sich das **DO** von Zeile 4 diese beiden Werte vom Stack und pusht sie auf den Return-Stack. Als nächstes kommt der **DUP**-Befehl an die Reihe. An dieser Stelle duplizieren wir somit das Quadrat des Index der äußeren Schleife. Dann lassen wir uns durch **I** den Index der mittleren Schleife auf den Stack legen, den wir mittels **SQUARE** quadrieren. Das letzte Wort auf Zeile 4, der Befehl **+**, sorgt dafür, daß die Zahl an oberster Stack-Position sich nun aus dem Quadrat der Indizes der äußeren und mittleren Schleife ergibt. Dies ist aber nichts anderes als $a^2 + b^2$, wobei a den Index der äußeren und b den Index der mittleren Schleife darstellt.

```
0 ( Pythagoreische Zahlentripel )
1 : SQUARE   DUP   *   ;
2 : PT 15 .R   ;
3 : PYTHTRIP 100   1   DO   I   DUP   SQUARE
4                100   I   DO   DUP   I   SQUARE   +
5                  142   I   DO   DUP   I   SQUARE   -   DUP
6                     0= IF I J 6 PICK   CR  PT   PT   PT  THEN
7                     0<   IF   LEAVE   THEN
8                  LOOP           DROP
9                LOOP          DROP      DROP
10            LOOP    CR       ;
11
12
13
14
15
```

ABBILDUNG 4-7: Ein Programm zur Berechnung pythagoreischer Zahlentripel zwischen 1 und 100

Jetzt kommt die innerste Schleife an die Reihe. Auch hier ist es wichtig, daß der Stack nach jedem Durchlauf durch diese Schleife unverändert bleibt. Zeile 5 gibt als erstes den Testwert 142 auf den Stack. Falls nämlich a und b gleich 100 oder kleiner sind, dann kann c, der Index der inneren Schleife, nicht größer als 142 sein (denn 142^2 ist größer als $100^2 + 100^2$). Als nächstes liefert uns **I** den Index der mittleren Schleife, der als Anfangswert für den Laufindex der inneren Schleife dient, während 142 der Testwert dieser Schleife ist. Den Grund, als Testwert für die innerste Schleife 142 zu nehmen, haben wir bereits dargestellt. Daß wir den Index der mittleren Schleife als Anfangswert für den Laufindex der innersten Schleife setzen, liegt daran, daß c nicht kleiner als a oder b sein kann. Die innerste Schleife soll ja alle möglichen Werte für c testen, und unmögliche Werte wollen wir dabei gar nicht erst betrachten. Das innerste **DO** in Zeile 5 entfernt diese Schleifenparameter und pusht sie auf den Return-Stack. Jetzt ist wieder $a^2 + b^2$ oberstes Stack-Element, welches mittels **DUP** dupliziert wird. Wir besorgen uns mit **I** den Index der inneren Schleife, quadrieren ihn mit **SQUARE** und subtrahieren diesen Wert vom zweiten Stack-Eintrag, so daß nach Ausführung von - der Stack zuoberst das Ergebnis von $a^2 + b^2 - c^2$ enthält. Falls diese Zahl gleich 0 ist, dann haben wir ein pythagoreisches Zahlentripel gefunden.

Diesen Test nehmen wir in Zeile 6 vor. Das Wort **0 =** liefert ein Flag, das wir mit **IF** auswerten. Falls es wahr ist, bedeutet dies, daß wir eines der gesuchten Tripel gefunden haben. Dieses wollen wir natürlich auf dem Bildschirm ausgeben. Die Wörter **I** und **J** hinter dem **IF**-Befehl legen die Laufindizes der innersten und mittleren Schleife auf den Stack. Der **PICK**-Befehl sorgt zusammen mit der Zahl 6, die zuvor auf den Stack gelegt wird, dafür, daß der Index der äußersten Schleife nach oben befördert wird. Wir haben somit alle 3 gewünschten Indizes an den richtigen Stellen und können sie mit unserem selbstdefinierten **PT** ausdrucken lassen. Wenn wir allerdings keines der gesuchten Zahlentripel gefunden haben, dann war das Flag, das in Zeile 6 mittels **0 =** getestet wurde, falsch, und die Befehle zwischen **IF** und **THEN** werden ignoriert.

Wenn das Quadrat des innersten Schleifenindex größer wird als die Summe der Quadrate der beiden äußeren Schleifenindizes, dann gibt es keinen Grund, die innerste Schleife noch weiter zu durchlaufen. Dieser Sachverhalt wird in Zeile 7 überprüft. Falls das Wort **0<** ein wahres Flag ergibt, dann verlassen wir mittels **LEAVE** die innerste Schleife. Wie Sie bereits wissen, erzwingt **LEAVE**, daß der Testwert der gerade in Ausführung befindlichen Schleife - in diesem Fall der innersten - gleich dem Schleifenindex gemacht wird. Das führt dazu, daß bei der Erreichung des **LOOP** diese Schleife (vorzeitig) verlassen wird. Wir befinden uns aber immer noch in der mittleren Schleife. Ehe wir jedoch einen neuen Durchgang durch die mittlere Schleife in Angriff nehmen können, müssen wir den Stack wieder in den Zustand überführen, in dem er sich vor Eintritt in die innerste Schleife befand. Nun haben wir in Zeile 4 mittels **DUP** den Wert von $a^2 + b^2$ auf den Stack dupliziert; diesen müssen wir jetzt wieder loswerden. Dafür sorgt das Wort **DROP** in Zeile 8. Wenn FORTH nun auf das Wort **LOOP** in Zeile 9 stößt, dann unternimmt es einen weiteren Durchgang durch die mittlere Schleife, vorausgesetzt deren Endekriterium ist noch nicht erreicht. Wir wiederholen diesen ganzen Prozeß und testen dabei immer wieder den Index der innersten Schleife, bis die mittlere Schleife insgesamt 100mal durchlaufen wurde. Danach, also nach Beendigung der mittleren Schleife, müssen wir den Index der äußeren Schleife und dessen Quadrat vom Stack entfernen, um alles für einen erneuten Durchgang durch die äußerste Schleife vorzubereiten. Dazu dienen die beiden **DROP** von Zeile 10. Danach geht's erneut in die äußerste Schleife, und der ganze Vorgang wiederholt sich erneut.

Sicher ist Ihnen aufgefallen, daß wir beim "Layout" unseres Programms darauf geachtet haben, jede Schleife eingerückt im Verhältnis zur vorherigen darzustellen. Dadurch wird das Programm übersichtlicher, denn Sie sehen sofort, welche Schleife in welche eingeschachtelt ist. Lesbarere Programme sind aber leichter zu verfolgen und entsprechend fehlerfrei zu machen.

Noch ein weiterer wichtiger Aspekt läßt sich an diesem Beispiel nachvollziehen. Wenn Sie Schleifen programmieren, dann sollten Sie darauf achten, daß der Stack nicht mit Daten überladen wird. Dies kann dazu führen, daß Sie unwissentlich Ihr Betriebssystem überschreiben, wodurch sich Ihr Rechner "totstellt" und nurmehr durch einen Kaltstart "wiederbelebt" werden kann. Ferner sollten Sie darauf achten, keine sog. Endlosschleifen zu schreiben, also Schleifen, die niemals abbrechen. So etwas ergibt sich etwa dann, wenn das Inkrement in einer Schleife den Wert 0 hat. Bei jedem Durchgang durch die Schleife würde 0 auf den Schleifenindex addiert werden, wodurch sich dieser natürlich niemals ändert und der Testwert auch nie erreicht wird. Eine andere Möglichkeit für Endlosschleifen ergibt sich, wenn sowohl der Anfangs- als auch der Testwert positiv, das Inkrement aber negativ ist.

4.6 Bedingte Schleifen

Wir können jetzt mit **DO** und **LOOP** Schleifen programmieren. Bei diesen Schleifen liegt von Anfang an fest, wie oft sie durchlaufen werden (durch Angabe des Testwerts und des Anfangswerts für den Laufindex), und es gibt keine elegante Methode, sie vorzeitig zu verlassen. Man kann zwar in den Schleifenkörper ein **IF** mit einem **LEAVE** einbauen (vergleiche das letzte Programmbeispiel), doch ist diese Methode nicht besonders übersichtlich, und es gibt - wie wir sehen werden - dafür elegantere Lösungen. Schleifen vom bisher besprochenen Typus, bei denen "unerbittlich" eine festgesetzte Anzahl von Schleifendurchgängen ausgeführt werden, nennt man unbedingte Schleifen. Eine andere Art von Schleifen, die sog. bedingten Schleifen, erlaubt es, auf komfortable Weise Bedingungen für ein vorzeitiges Verlassen der Schleife mit anzugeben. In diesem Abschnitt werden wir einige Möglichkeiten zum Programmieren von solchen bedingten Schleifen kennenlernen.

BEGIN und **UNTIL** - Die FORTH-Wörter **BEGIN** und **UNTIL** dienen zum Programmieren von bedingten Schleifen. Mit diesen beiden Wörtern gebildete bedingte Schleifen haben die folgende allgemeine Form:

BEGIN anweisungen a (flag) UNTIL (4-22)

Dies funktioniert wie folgt. Wenn der FORTH-Interpreter zum ersten Mal auf das Wort **BEGIN** trifft, dann führt er die "anweisungen a" aus. Beim Auftauchen von **UNTIL** wird der oberste Stack-Eintrag entfernt und als Flag behandelt. Dieses Flag kommt in der Regel durch Operationen innerhalb der "anweisungen a" auf den Stack. Wenn das Flag falsch ist, dann werden die "anweisungen a" wiederholt. Dies geht so lange weiter, bis das Flag wahr wird. In diesem Fall wird die Schleife verlassen, und FORTH führt die Worte hinter **UNTIL** aus. Das Wort **UNTIL** hat folgende Stack-Relation:

n -> (4-23)

Als Beispiel für eine bedingte Schleife wollen wir ein FORTH-Wort schreiben, das eine beliebige Anzahl von Zahlen aufaddiert. Wenn wir das Wort **ADDER** aufrufen, dann fordert es den Benutzer zur Eingabe einer Zahl auf, indem es ein Fragezeichen auf den Bildschirm bringt. Nach Eingeben einer Zahl (und Drücken der RETURN-Taste) erscheint erneut das "?". Auf diese Art können Sie eine beliebige Anzahl von Zahlen eingeben; die Zahleneingabe wird durch eine Null beendet. Wenn das Programm bemerkt, daß Sie 0 eingegeben haben, dann bricht es die Schleife ab und gibt die Summe aller bisher eingegebenen Zahlen aus. Sie sehen das Programm in Abb. 4-8.

```
0 ( Summe einer beliebigen Anzahl von Zahlen )
1 : ADDER   0   BEGIN   #IN   DUP   ROT   +  SWAP
2               0 =   UNTIL   CR    .    ;
3
4
5
```

ABBILDUNG 4-8: Ein Programm zum Aufaddieren einer beliebigen Anzahl von Zahlen

Wenden wir uns nun den Einzelheiten dieses Programms zu. Als erstes pushen wir 0 auf den Stack. Dann wird durch **BEGIN** die bedingte Schleife eingeleitet. Die erste Anweisung innerhalb der Schleife ist das Wort **#IN**, welches die Eingabeaufforderung ? auf den Bildschirm bringt und so lange wartet, bis der Benutzer eine Zahl eingegeben hat. Diese Zahl duplizieren wir. Dann holen wir uns mit **ROT** den dritten Stack-Eintrag an die oberste Position; bei Eintritt in die Schleife ist dies eine 0, bei den nachfolgenden Schleifendurchgängen befindet sich an dieser Stelle jedoch die Summe der bisher eingegebenen Zahlen. Mittels + addieren wir dazu die letzte Benutzereingabe. Diese Zahl wird anschließend von **SWAP** an die oberste Stack-Position befördert. Wir testen nun mit **0** =, ob es sich dabei um eine 0 handelt. Bekanntermaßen entfernt **0** = die Testzahl und legt statt ihrer ein Flag auf den Stack. Dieses ist nur dann wahr, wenn die zum Vergleich herangezogene Zahl gleich 0 war. Das **0** = ist auch Grundlage für die Entscheidung von **UNTIL**. Ist es falsch, dann erfolgt ein erneuter Durchgang durch die Schleife, was bedeutet, daß die Befehle zwischen **BEGIN** und **UNTIL** erneut durchlaufen werden. Wir haben die Schleife so geschrieben, daß am Ende eines Schleifendurchlaufs die Summe der bisher eingegebenen Zahlen an oberster Stack-Position steht. Hat nun der Bediener zuletzt eine 0 eingegeben, dann ist das von **0** = erzeugte Flag in Zeile 2 wahr. In diesem Fall wird die Schleife verlassen, wir erzeugen mittels **CR** einen Zeilenvorschub und geben die bisher berechnete Summe aus.

BEGIN, **WHILE** und **REPEAT** - Mit diesen drei Worten kann in FORTH eine andere Art von bedingter Schleife formuliert werden. Diese hat die allgemeine Form:

BEGIN anweisungen a (flag) WHILE anweisungen b REPEAT (4-24)

Eine mit **BEGIN**, **WHILE** und **REPEAT** gebildete Schleife funktioniert folgendermaßen. Der Anfang der Schleife wird durch **BEGIN** eingeleitet. Zuerst einmal werden die "anweisungen a" ausgeführt. **WHILE**, welches hinter diesen Anweisungen kommt, behandelt den obersten Stack-Eintrag als Flag und entfernt ihn von dort. Falls das Flag wahr ist, dann sorgt **WHILE** dafür, daß die "anweisungen b" *ausgeführt werden und anschließend zum Anfang der Schleife* unmittelbar hinter **BEGIN** zurückgekehrt wird. Es werden in diesem Fall also auch die "anweisungen a" erneut ausgeführt. Ist aber das Flag, das durch **WHILE** getestet wird falsch, dann wird die

Schleife verlassen, und die "anweisungen b" gelangen nicht zur Ausführung. Vielmehr werden die Befehle ausgeführt, die sich an das Wort **REPEAT** anschließen. Diese Arbeitsweise sorgt dafür, daß die mit **BEGIN-WHILE-REPEAT** gebildete Schleife so lange ausgeführt wird, solange eine Bedingung wahr ist; außerdem kann diese Bedingung an beliebiger Stelle innerhalb der Schleife deren Verlassen erzwingen. Andererseits wird eine **BEGIN-UNTIL**-Schleife so lange ausgeführt, solange eine Bedingung falsch ist, und diese Schleife kann nur am Ende verlassen werden.

Als Beispiel wollen wir das Wort zur Berechnung der Fakultät einer Zahl (vergleiche Abschnitt 4-3) neu definieren. Diese Alternative zu unserer bereits bekannten Definition sehen Sie in Abb. 4-9.

```
0  ( Alternative Fakultätsdefinition )
1  : FACT   DUP   BEGIN   1-   DUP   1-   0>   WHILE
2           DUP   ROT   *   SWAP   REPEAT
3           SWAP  .     DROP   ;
4
5
6
7
8
9
10
12
13
14
15
```

ABBILDUNG 4-9: Alternative Definition der Fakultät

Wie bereits zuvor (4-15) geben wir unserem Wort den Namen **FACT**. Nun zu den Einzelheiten dieser Definition: Bei Aufruf von **FACT** sollte sich eine Zahl auf dem Stack befinden, deren Fakultät berechnet werden soll. Diese Zahl wird zuerst in Zeile 1 dupliziert, ehe wir in die Schleife eintreten. Innerhalb der Schleife wird die Zahl als erstes dekrementiert, wir ziehen mittels **1-** 1 von der Zahl ab. Das Wort **DUP** dupliziert diese dekrementierte Zahl, wonach wir sie mit **1-** erneut dekrementieren. Das Ergebnis

dieses "Zurückzählens" testen wir nun mit **0>**. Dieses Wort legt bekanntermaßen ein wahres Flag auf den Stack, wenn dessen oberster Eintrag größer Null war. **WHILE** entfernt dieses Flag; falls es wahr ist, geschieht nichts weiter. FORTH fährt einfach mit der Schleife fort und führt das **DUP** aus. Der Stack enthält nun die Originalzahl, welche wir n nennen wollen und darüber zweimal diese Originalzahl um 1 vermindert. Mit **ROT** wird die Originalzahl an oberste Stack-Position gebracht. Jetzt berechnet das FORTH-Wort ***** das Produkt n*(n-1) und legt es auf den Stack. Nach dem **SWAP** befindet sich n-1 an oberster Stack-Position und n*(n-1) an zweiter Stelle. Danach erfolgt ein Wiedereintritt in die Schleife. Der oberste Stack-Eintrag wird um 1 vermindert. Nach dem zweiten Durchgang durch die Schleife haben wir somit n*(n-1)*(n-2) berechnet. In dieser Art geht das Programm weiter, bis der zweite **1-**-Befehl in Zeile 1 die Zahl auf 0 reduziert. In diesem Fall ist das Flag falsch. Stößt aber **WHILE** auf ein falsches Flag, dann übergeht es den Rest der Schleife, also die Anweisungen zwischen **WHILE** und **REPEAT**. Die Schleife wird beendet. Mit **SWAP** in Zeile 3 kommt die gewünschte Fakultät an die oberste Stack-Position und wird anschließend durch den Punktbefehl ausgegeben. Ehe wir die Definition des Wortes beenden, bereinigen wir noch mit **DROP** den Stack.

Bedingte Schleifen hören nur dann mit der Arbeit auf, wenn eine bestimmte Bedingung erfüllt ist. Wenn sich in Ihrem Programm ein Fehler befindet, dann kann es sein, daß diese Bedingung nie eintritt und eine Endlosschleife zustande kommt. Wenn beim Austesten eines selbstdefinierten Wortes der Computer "einzuschlafen" scheint, dann kann durchaus eine solche Endlosschleife dafür verantwortlich sein. Überprüfen Sie Ihre Programme also sorgfältig, um es nicht soweit kommen zu lassen.

4.7 Einige zusätzliche Vergleichswörter

FORTH stellt noch einige weitere Vergleichsworte zur Verfügung, die wir in diesem Abschnitt erörtern wollen. Da wir die grundlegende Funktionsweise von Vergleichswörtern bereits kennengelernt haben, werden wir die neuen Wörter nur kurz vorstellen.

?DUP - Dieses Kommando dupliziert die oberste Zahl auf dem Stack, vorausgesetzt, diese ist ungleich 0. Ist der oberste Stack-Eintrag aber gleich 0, dann macht **?DUP** nichts.

ABORT - Das FORTH-Kommando **ABORT** bereinigt sowohl den Datenstack als auch den Return-Stack, beendet das Programm und übergibt die Kontrolle an das Terminal. Man kann damit also sowohl den Stack bereinigen als auch ein Programm beenden. Einige FORTH-Systeme verwenden für diesen Zweck ein etwas anderes Kommando, nämlich **ABORT"**. Man benutzt es in der Form

ABORT" text " (4-25)

Die Funktionsweise von **ABORT"** ist ähnlich wie die von **ABORT**, außer daß dieses Wort aufgrund eines Flags in Aktion tritt. Wenn das Flag an oberster Stack-Position wahr ist, dann löscht **ABORT"** ebenfalls die beiden Stacks und beendet das Programm, druckt aber zusätzlich noch das Textmaterial zwischen den Anführungszeichen auf dem Terminal aus. Man kann mit **ABORT"** also die Programmausführung abbrechen, um irgendwelche ungewünschten Effekte zu verhindern. Ist das Flag falsch, dann beseitigt **ABORT"** einfach das Flag und hat ansonsten keine Wirkung.

Wir wollen ein kleines Beispiel für den Einsatz von **ABORT"** geben. Stellen Sie sich vor, daß Sie eine Schleife geschrieben haben, deren Schleifeninkrement entweder im Programm berechnet oder vom Benutzer eingegeben wird. Ergibt sich nun (entweder durch Berechnung oder durch Benutzereingabe) ein Inkrement von 0, so würde dies zu einer der gefürchteten Endlosschleifen führen. Sie können diesen Fall aber mit **ABORT"** abfangen. Betrachten Sie dazu den folgenden Ausschnitt aus einem FORTH-Wort:

```
DUP 0= ABORT" PROGRAMM BEENDET " +LOOP
```

Der oberste Stack-Eintrag wird dupliziert und anschließend mit **0=** verglichen. Wir haben jetzt ein Flag auf dem Stack stehen, das von **ABORT"** beseitigt wird. Falls es falsch ist, dann hat **ABORT"** keine Wirkung. Ist das Flag andererseits wahr, dann wird das Programm beendet, und wir erhalten die Meldung PROGRAMM BEENDET auf dem Bildschirm.

Y/N - Dieses Wort ist kein Teil von FORTH-79, ist jedoch in MMS-FORTH implementiert und wird deshalb hier dargestellt. Bei Ausführung von **Y/N** unterbricht das Programm seine Arbeit und bringt die Meldung (Y/N)? auf den Bildschirm. Der Benutzer muß nun entweder mit Y (für englisch "yes" = ja) oder N (für "nein") antworten. Um diese Antworten in den Computer einzugeben, braucht die RETURN-Taste nicht betätigt zu werden. Nach Eingabe eines dieser beiden Zeichen wird ein Flag auf den Stack gelegt. Dieses ist falsch, wenn der Benutzer die Y-Taste betätigt hat und bei Eingabe von N wahr. Das Wort verhält sich somit genau anders herum, als man es erwarten würde; gehen Sie deshalb damit vorsichtig um! Seine Stack-Relation ist:

$$\rightarrow n_{flag} \qquad (4\text{-}26)$$

Sie können also mit **Y/N** dem Benutzer Entscheidungsfragen (Ja-Nein-Fragen) stellen.

Die Abbildung 4-10 liefert ein Beispiel für den Einsatz von **Y/N**. Wir definieren darin ein Wort mit dem Namen **PTZ2**. Dieses Programm druckt die erste Potenz von 2 (also die 2) und fragt dann den Benutzer, ob es weitermachen soll. Antwortet dieser mit Y, dann wird die nächste Zweier-Potenz (4) gedruckt. Das Programm macht damit so lange weiter, bis der Benutzer auf das Fragezeichen des Programms mit einem N antwortet. Sehen wir uns Abb. 4-10 an. Die Zweierpotenzen werden in einer **BEGIN-UNTIL**-Schleife berechnet und ausgegeben. Vor dem **UNTIL** erfolgt jedoch ein Aufruf von **Y/N**. Antworten wir jetzt mit Y, dann wird ein Flag mit dem Wahrheitswert falsch auf den Stack gelegt und somit die Schleife erneut durchlaufen. Reagiert der Benutzer auf die Nachfrage des Systems mit N, dann findet **UNTIL** ein wahres Flag auf dem Stack vor und bricht die Schleife ab.

```
0 ( EIN BEISPIEL FüR Y/N )
1 : PTZ2  1   BEGIN   2   *   DUP   CR
2                " WOLLEN SIE WEITERMACHEN "   CR
3                Y/N   CR   UNTIL          ;
4
```

ABBILDUNG 4-10: Programm, das vom Benutzer abgebrochen wird

NOT - Das FORTH-Kommando **NOT** entfernt den obersten Stack-Eintrag und betrachtet ihn als Flag. Ist das Flag wahr, dann legt es ein falsches Flag auf den Stack. Ist das Ausgangsflag aber falsch, dann wird ein wahres Flag auf den Stack gelegt. Die Stack-Relation von **NOT** lautet

$$n_{flag1} \text{ --> } n_{flag2} \quad (4\text{-}27)$$

Wenn wir etwa dem **Y/N**-Befehl das Wort **NOT** folgen lassen, dann haben wir als Ergebnis ein wahres Flag, wenn der Benutzer Y eingegeben hat. Bei Eingabe von N ist das Flag hingegen falsch.

4.8 Verzweigung mittels CASE

In diesem Abschnitt werden Sie ein weiteres Verfahren kennenlernen, mit dem man Verzweigungen in FORTH-Programmen erreichen kann. Hierbei handelt es sich um keine in FORTH-79 vorgesehene Möglichkeit, sie ist jedoch in MMSFORTH implementiert. Es wäre sehr komfortabel, wenn man in einem Programm zu verschiedenen FORTH-Wörtern verzweigen könnte. Wir könnten z.B. wünschen, unter einer Bedingung ein FORTH-Wort auszuführen, während unter einer anderen Bedingung ein anderes zur Ausführung gelangen soll. Diesen Fall nennt man Verzweigung durch Auswahl. Diese Verzweigungsmöglichkeit wird durch das FORTH-Wort **NCASE** zur Verfügung gestellt. Programme mit **NCASE** haben folgende allgemeine Form:

NCASE n_1 n_2 n_3 " WORTA WORTB WORTC
OTHERWISE anweisungen q CASEND anweisungen x (4-28)

Hierbei stellen n_1, n_2 und n_3 eine Liste von Integers zwischen -128 und 127 dar. An diese schließt sich ein einzelnes " sowie eine Liste von FORTH-Wörtern an. Im Beispiel (4-28) haben wir drei Zahlen und drei FORTH-Wörter, es können aber genausogut mehr Zahlen und Wörter bzw. weniger Zahlen und Wörter sein. Die Anzahl der Zahlen vor und der Wörter nach dem Anführungszeichen muß jedoch gleich sein. An die Liste der FORTH-Wörter schließt sich

das Schlüsselwort **OTHERWISE** an. Dahinter können beliebige FORTH-Ausdrücke stehen. Anschließend geben Sie das Wort **CASEND** ein.

Der ganze Ausdruck funktioniert folgendermaßen: Die Zahlenliste korrespondiert mit der Liste von FORTH-Wörtern. Bei Ausführung von **NCASE** wird die oberste Zahl vom Stack entfernt. Falls Sie gleich n_1 ist, dann wird "WORTA" ausgeführt. Ist die Zahl gleich n_2, dann kommt "WORTB" an die Reihe. Nur eines der Wörter aus der Liste kommt zur Ausführung; die "anweisungen q" werden nicht ausgeführt. Als nächstes werden - falls vorhanden - die "anweisungen x" hinter dem **CASEND** ausgeführt. Entspricht die Zahl, die vor Ausführung von **NCASE** auf dem Stack war, keiner der Zahlen der Liste, dann wird auch keines der Wörter in der Liste hinter dem " aufgerufen. Statt dessen werden die "anweisungen q" abgearbeitet. Danach schließen sich die "anweisungen x" hinter dem **CASEND** an. **NCASE** besitzt folgende Stack-Relation:

n --> (4-29)

In Abbildung 4-11 sehen Sie ein Beispiel für den Einsatz von **NCASE.** Hierbei handelt es sich um ein Programm, das die zweite und dritte Zahl auf dem Stack miteinander addiert, subtrahiert oder multipliziert. Welche dieser drei Operationen ausgeführt wird, bestimmt die erste Zahl auf dem Stack. Ist der oberste Stack-Eintrag gleich 1, dann wird addiert; handelt es sich dabei um eine 4, dann wird subtrahiert. Ähnlich sorgt eine 7 für Multiplikation. (In der Zahlenliste innerhalb von **NCASE** müssen die Zahlen nicht in numerischer Reihenfolge stehen. Die erste Zahl der Zahlenliste entspricht dem ersten Wort in der Wörterliste, die zweite Zahl in der Zahlenliste entspricht dem zweiten Wort in der Wörterliste usw.)

Betrachten wir nun das Programm. In den ersten drei Zeilen definieren wir die neuen Wörter **ADDIERE, SUBTRAHIERE** und **MULTIPLIZIERE,** die jeweils 2 Zahlen vom Stack entfernen, die angegebene Operation ausführen und deren Ergebnis auf den Bildschirm bringen. In Zeile 4 beginnen wir die Definition von **AUSWAHL.** Bei Ausführung von **NCASE** wird die oberste Zahl vom Stack entfernt. Handelt es sich um eine 1, dann wird das Wort **ADDIERE** aufgerufen, worauf sich die Anweisungen hinter **CASEND** anschließen. Wurde eine 7 vom Stack entfernt, dann wird statt dessen das Wort **MULTIPLIZIERE** ausgeführt. Ist die auf dem Stack befindliche Zahl jedoch

weder eine 1 noch eine 4 oder 7, dann gelangt keines der Wörter zwischen dem Anführungszeichen und **OTHERWISE** zur Ausführung; statt dessen werden in diesem Fall die Anweisungen zwischen **OTHERWISE** und **CASEND** bearbeitet. Das bedeutet, daß wir die Meldung "FALSCHE AUSWAHL" geben. Die Befehle, die hinter **CASEND** stehen, gelangen also auf jeden Fall zur Ausführung. Das bedeutet, daß das Programm stets mit der Meldung "PROGRAMM BEENDET" aufhört.

```
0 ( EIN BEISPIEL FüR CASE )
1 : ADDIERE    +    .   ;
2 : SUBTRAHIERE   -    .   ;
3 : MULTIPLIZIERE   *    .   ;
4 : AUSWAHL NCASE   1 4 7   " ADDIERE SUBTRAHIERE MULTIPLIZIERE
5      OTHERWISE   " FALSCHE AUSWAHL "     CASEND
6          . " PROGRAMM BEENDET "     ;
7
8
9
10
12
13
14
15
```

ABBILDUNG 4-11: Ein Beispiel für den Einsatz von **NCASE**

4.9 Strukturierte Programmierung

Ein Programm ist dann strukturiert, wenn es in kleine Einheiten (sog. Moduln) gegliedert ist, die leicht zu verstehen und zu verbessern sind. FORTH bietet sich für die strukturierte Programmierung geradezu an, denn die meisten Programme werden fast automatisch in kleine selbständige Untereinheiten - die oben erwähnten Module - aufgeteilt. In diesem Abschnitt stellen wir genauer dar, wie man strukturierte Programme schreibt.

4.9.1 Modularisierung

Ein FORTH-Programm besteht in der Regel aus einer Vielzahl von Kommandos. So haben wir beispielsweise in der Abbildung 4-7 innerhalb des selbstdefinierten Wortes **PYTHTRIP**, welches das Hauptprogramm ist, einen Aufruf der selbstdefinierten Wörter **SQUARE** und **PT**. Wenn es sich um ein sehr kompliziertes Programm handeln würde, dann könnte das Hauptprogramm eine Vielzahl anderer selbstdefinierter Wörter aufrufen. Diese aber können ihrerseits wieder andere Wörter rufen. Im allgemeinen sollten Sie versuchen, Ihre Programme so kurz wie möglich zu halten. Als Faustregel für die Länge von Programmen gilt, daß sie keinesfalls die Größe eines Bildschirms überschreiten sollten. Auf diese Art halten Sie jedes Wort möglichst einfach, so daß Sie es leicht testen und fehlerfrei machen können. Kompliziertere Programme werden dann mit Hilfe bereits entwickelter und getesteter Wörter aufgebaut. Dieses Verfahren macht auch die Programmierung im Team möglich; verschiedene Programmierer können so an unterschiedlichen Worten arbeiten. Natürlich können Sie mit der Arbeit an einfachen Worten so lange nicht beginnen, solange nicht die gesamte Struktur des Programms festgelegt ist. Dieses muß der erste Arbeitsschritt sein.

4.9.2 Algorithmen - Programmentwicklung

Wenn Sie ein Programm schreiben - egal, ob einfach oder kompliziert - dann müssen Sie sich dafür einen allgemeinen Plan zurechtlegen. Dieser Plan, der die allgemeine Vorgehensweise beschreibt, wird auch als Algorithmus bezeichnet. Ehe Sie sich an die tatsächliche Programmierarbeit machen, müssen Sie sich also über den zu verwendenden Algorithmus klar werden. Falls das Programm einfach ist, dann mag es erscheinen, als ob dieser erste Schritt übergangen und das Programm gleich geschrieben werden könnte. In diesen Fällen hat der Programmierer den Algorithmus bereits im Kopf. Bei komplizierteren Programmen ist dieses Verfahren unmöglich. Es sollte dann auf jeden Fall erst einmal eine "Ideensammlung" angelegt werden, die als Grundlage für die Verfeinerung des Algorithmus dienen kann. Erinnern wir uns an das Problem mit dem pythagoreischen Zahlentripel. Die erste Idee, die ein Programmierer zu diesem Problem haben könnte, besteht darin,

drei Schleifen ineinander zu schachteln. Die beiden äußeren Schleifen sollen der Reihe nach alle Werte für a^2 und b^2 generieren, während die innere Schleife die Werte für c^2 berechnet und diese mit der Summe von a^2 und b^2 vergleicht. Wenn die beiden Größen gleich sind, dann haben wir ein pythagoreisches Zahlentripel gefunden. Dies ist auch die grundlegende Idee, nach der das Beispiel 4-7 konstruiert ist. Natürlich ist der Algorithmus jetzt noch nicht vollständig. Es gibt noch eine Menge von Details, die festgelegt werden müssen. Auch bei einem sehr komplizierten Programm würde man ähnlich vorgehen: zuerst skizziert man das allgemeine Verfahren. Letztendlich wird dieses allgemeine Verfahren dann in Form eines FORTH-Wortes implementiert. In diesem Wort kann es eine Vielzahl von Unterworten geben. Solange wir uns noch am Anfang der Programmentwicklung befinden, legen wir nur fest, was diese Unterworte leisten sollen, belasten uns aber noch nicht mit der Frage, wie dies erreicht werden kann. Erst nachdem wir das Hauptwort geschrieben und fehlerfrei gemacht haben, wenden wir uns den einzelnen Unterworten zu und wiederholen dabei dieses Vorgehen. Wir entwickeln also für jedes Unterwort einen eigenen Algorithmus, welcher auch auf weitere Unterwörter zurückgreifen kann. Wir entwickeln und testen erst einmal diesen Algorithmus, ehe wir uns den Unterworten der nächsten Ebene zuwenden.

Diesen Sachverhalt kann man in einem hierarchischen Diagramm darstellen. Ein Beispiel dafür sehen Sie in Abbildung 4-12. In dieser Abbildung ruft das oberste FORTH-Wort zwei weitere Unterworte, a und c. Diese Unterwörter rufen ihrerseits die Unterwörter b, d und e. Dabei greifen sowohl Unterwort a als auch Unterwort c auf Unterwort d zu. Erst wenn die Beziehung der einzelnen Wörter zueinander klar ist und wir uns vergewissert haben, daß unser Algorithmus fehlerfrei ist, sollten wir anfangen, die Wörter tatsächlich in FORTH zu implementieren.

Die eben geschilderte Vorgehensweise nennt man Programmentwicklung von oben nach unten (Top-down design). Sie erscheint dem Anfänger oftmals paradox. Man beginnt mit dem hierarchischen Diagramm, aus dem man das oberste Wort zur Programmierung auswählt. Dies wird getestet, wobei jedoch noch keines seiner Unterwörter bereits definiert ist. Damit das Hauptwort trotzdem ausgetestet werden kann, versehen wir die benötigten Unterwörter mit Hilfsdefinitionen, sog. Programmrümpfen. Diese Rümpfe führen keine tatsächlichen Berechnungen aus; sie übergeben jedoch gegebenenfalls Testdaten an das Hauptprogramm, damit dieses richtig funktionieren kann.

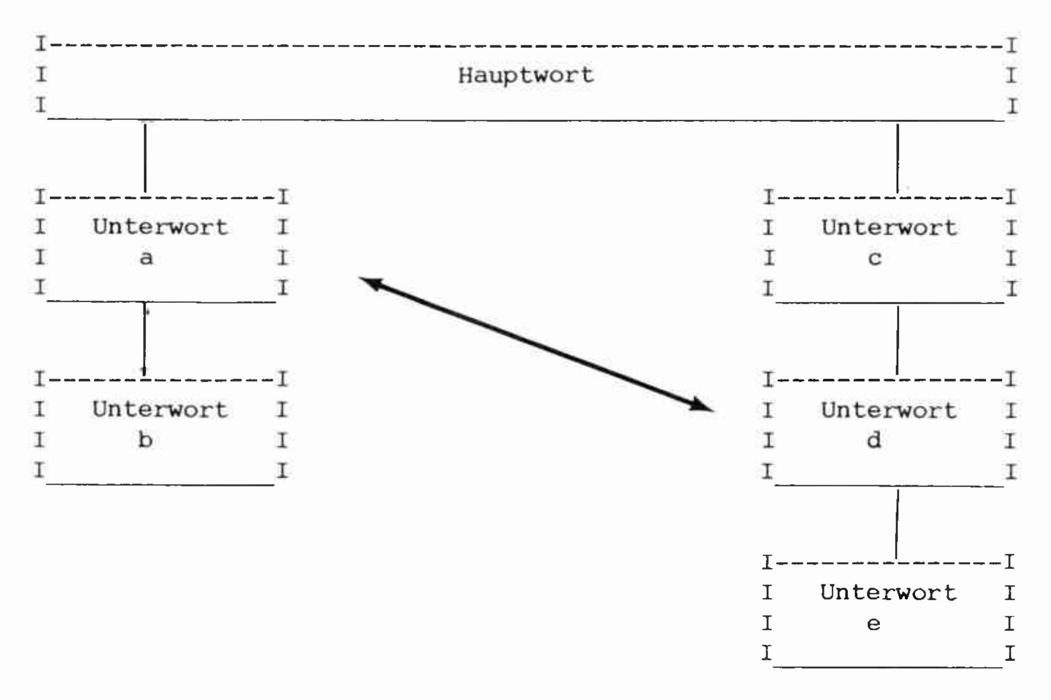


ABBILDUNG 4-12: Ein einfaches hierarchisches Diagramm

Nachdem wir das Hauptprogramm überprüft und uns vergewissert haben, daß es richtig funktioniert, arbeiten wir der Reihe nach die einzelnen Unterwörter aus, indem wir deren Programmrümpfe durch tatsächliche Definitionen ersetzen. Bei der Entwicklung eines Unterwortes gehen wir wieder nach der Top-down-Methode vor. Verwendet ein Unterwort seinerseits weitere Unterwörter, so schreiben wir für diese Unterwörter nur Behelfsdefinitionen (Programmrümpfe), um uns ganz der Lösung des gerade betrachteten Problems (des Unterwortes der ersten Ebene) widmen zu können. Noch einmal zu Abbildung 4-12. Als erstes entwickeln wir das Hauptwort, wobei wir für die Unterwörter a und c nur Programmrümpfe verwenden. Als nächstes entwickeln und testen wir Unterwort a, wobei wir Hilfsdefinitionen für die Unterwörter b und d verwenden. Nachdem Unterwort a ausgetestet ist, wenden wir uns dem Unterwort b zu. In dieser Art, also von oben nach unten durch das hierarchische Diagramm gehend, wird jedes einzelne Modul des komplexen Programms entwickelt und getestet. Das Verfahren erscheint Ihnen jetzt vielleicht umständlich; beim Entwickeln größerer Programme ist es jedoch eine große Hilfestellung.

4.10 Übungsaufgaben

In den folgenden Übungsaufgaben sollen Sie selbst FORTH-Wörter definieren bzw. FORTH-Programme schreiben. Überprüfen Sie Ihre Lösungen selbst, indem Sie sie auf Ihrem Computer austesten. Halten Sie die einzelnen FORTH-Wörter so kurz wie möglich und versuchen Sie, komplizierte Aufgaben zu lösen, indem Sie eine Reihe von Unterwörtern aus einem Hauptwort heraus aufrufen.

4-1 Was bedeutet der Ausdruck "Flag"?

4-2 Schreiben Sie ein FORTH-Wort, das zwei Zahlen subtrahiert und deren Differenz ausgibt, falls sie ungleich sind, ansonsten aber die Meldung DIE ZAHLEN SIND GLEICH auf den Bildschirm bringt.

4-3 Schreiben Sie ein FORTH-Wort, das die kleinste von drei Zahlen auf dem Stack ausgibt. Verwenden Sie dazu die Vergleichswörter aus Abschnitt 4-1.

4-4 Wiederholen Sie Aufgabe 4-3, geben Sie diesmal aber die größte von drei Zahlen auf dem Stack aus.

4-5 Wiederholen Sie Aufgabe 4-4, geben Sie diesmal aber den größten Absolutwert aus.

4-6 Was bedeutet der Ausdruck "bedingte Verzweigung"?

4-7 Schreiben Sie ein FORTH-Wort, das 3 Zahlen addiert und sie mit 4 multipliziert, wenn ihre Summe kleiner 50 ist. Wenn die Summe zwischen 50 und 70 liegt, dann sollen die Zahlen mit 6 multipliziert werden, ist die Summe größer als 70, dann multiplizieren Sie sie mit 8.

4-8 Schreiben Sie ein FORTH-Wort, das die folgende Polynomial-Gleichung für verschiedene Werte von x berechnet:

$$4x^3 + gx^2 + 5x + 3$$

dabei soll gelten g = 4, falls x kleiner gleich 15 ist, und g = 9 für x größer 15.

4-9 Schreiben Sie ein FORTH-Wort, das die Summe aller Zahlen berechnet, die zwischen zwei Integers liegen. Das Wort liest also zwei Einträge vom Stack, die den unteren und oberen Grenzwert für die Zahlenreihe angeben, und liefert die Summe der Zahlen innerhalb dieser Grenzen. Gehen Sie davon aus, daß der kleinere Grenzwert der oberste Stack-Eintrag ist.

4-10 Schreiben Sie ein FORTH-Wort zur Berechnung des Polynoms

$$x^3 - 2x^2 + 2x - 15$$

für alle Werte von x zwischen 1 und 20.

4-11 Schreiben Sie ein FORTH-Kommando, das die folgende Funktion für alle Werte von x zwischen -2 und 5 und für alle Werte von y zwischen -6 und 4 berechnet:

$$3xy^3 - 2x^2 + 5xy - 3x - 5y + 18$$

4-12 Wiederholen Sie Aufgabe 4-9, verlassen Sie diesmal aber das Wort mittels **LEAVE**, wenn die Summe größer als 10000 wird.

4-13 Schreiben Sie ein FORTH-Wort, das das Produkt aller ungeraden Zahlen zwischen 1 und einer ungeraden Zahl an oberster Stack-Position berechnet. Das Programm sollte abbrechen, wenn das Produkt größer 20000 wird. In diesem Fall sollten Sie eine passende Fehlermeldung ausgeben.

4-14 Wiederholen Sie Aufgabe 4-12, und verwenden Sie dabei die FORTH-Wörter **BEGIN-UNTIL**.

4-15 Wiederholen Sie Aufgabe 4-13, und verwenden Sie dabei die FORTH-Wörter **BEGIN-WHILE-REPEAT**.

4-16 Die Teilnehmer eines Kurses müssen jeweils 4 Prüfungen ablegen. Schreiben Sie ein modularisiertes FORTH-Wort (also eines, das seinerseits selbstdefinierte FORTH-Wörter ruft), welches folgende Berechnungen ausführt: Die Punktzahl in jeder der vier Prüfungen wird auf den Stack gelegt. Dann soll das Programm den Durchschnitt des einzelnen Studenten aus diesen vier Tests berechnen und in Abhängigkeit davon eine Bewertung ausgeben. Bei einem Durchschnitt von 90 oder besser erhält der Student die Bewertung A. Liegt der Schnitt

zwischen 89 und 80, so wird die Bewertung B vergeben. Mit einem Schnitt zwischen 79 und 70 erzielt man C, zwischen 69 und 60 D. Ist schließlich der Durchschnitt des Studenten schlechter als 59, dann geben Sie die Meldung DURCHGEFALLEN aus. Der berechnete Durchschnitt muß ganzzahlig sein. Wenn Sie die Division ausführen, sollten Sie den Quotienten um 1 erhöhen, falls der Divisionsrest 2 oder mehr beträgt.

4-17 Diskutieren Sie die Einsatzmöglichkeiten für das FORTH-Wort **ABORT**.

4-18 Vergleichen Sie die beiden FORTH-Wörter **ABORT** und **ABORT"**.

4-19 Wiederholen Sie Aufgabe 4-16, wobei diesmal das Programm nach beendeter Arbeit den Benutzer fragen soll, ob ein erneuter Durchgang gewünscht wird. Ändern Sie auch die Form der Dateneingabe. Das Programm soll seine Daten nicht auf dem Stack erwarten, sondern sie selbst vom Benutzer abfragen.

4-20 Wiederholen Sie Aufgabe 4-19, und verwenden Sie diesmal das FORTH-Wort **NOT**.

4-21 Schreiben Sie ein FORTH-Wort, das mit Hilfe anderer selbstdefinierter FORTH-Wörter für jeden Wochentag eine andere Meldung ausgibt. (Denken Sie sich diese Meldungen selbst aus.)

4-22 Was bedeutet der Ausdruck "strukturierte Programmierung"?

4-23 Was bedeutet der Ausdruck "Programmentwicklung von oben nach unten" (Top-down design)?

4-24 Wiederholen Sie die Aufgabe 4-19, wobei Sie diesmal darauf achten, bei der Programmentwicklung von oben nach unten vorzugehen.

5

Grundlegendes über Zahlen

5 Grundlegendes über Zahlen

Im überwiegenden Teil dieses Buches haben wir mit einfach genauen ganzen Zahlen (Integers) gearbeitet. Dieses Kapitel stellt das Arbeiten mit doppelt genauen Integers dar. Dabei können wir mit wesentlich größeren Zahlenbeträgen rechnen. Dies ist jedoch nicht der einzige Grund für die Einführung der doppelt genauen Integers. Wir werden nämlich in diesem Kapitel ein Verfahren, die sog. Skalierung kennenlernen, das es uns erlaubt, Integerarithmetik dort einzusetzen, wo normalerweise mit Bruchzahlen gerechnet wird. Weiterhin werden wir noch andere Formen der Integerarithmetik kennenlernen.

Manchmal müssen wir mit sehr großen oder sehr kleinen Zahlen arbeiten. In diesen Fällen ist oftmals selbst die Arithmetik mit doppelt genauen Zahlen nicht ausreichend, und wir benötigen sog. Gleitkommazahlen. Diesen Zahlentyp werden wir im vorliegenden Kapitel vorstellen. Gleitkommaarithmetik ist zwar kein Bestandteil von FORTH-79, findet sich jedoch in vielen anderen FORTH-Systemen und ist deshalb Gegenstand dieses Kapitels. Zwar benötigt die Gleitkommaarithmetik mehr Rechenzeit und Speicherplatz als die Arithmetik mit Integers, seien sie einfach oder doppelt genau, doch oftmals ist sie ganz praktisch. Wir werden deshalb in diesem Kapitel verschiedene Formen des Rechnens mit Gleitkommazahlen darstellen.

5.1 Doppelt genaue Integers

Einfach genaue Integers beanspruchen eine Stack-Position. Wie wir bereits ausgeführt haben, kann man mit diesem Datentyp Zahlen zwischen -32678 und 32767 darstellen. Weil doppelt genaue Integers in zwei Stack-Positionen gespeichert werden, sind damit Zahlen zwischen -2147483648 und 2147483648 möglich. Im Abschnitt 2-5 haben wir bereits einige Details der Stack-Manipulation im Zusammenhang mit doppelt genauen Integers dargestellt. Jetzt wenden wir uns den Einzelheiten des Rechnens mit diesen Zahlen zu. Man sollte doppelt genaue Zahlen nur dann verwenden, wenn ihr großer Wertevorrat unbedingt benötigt wird. Sie brauchen nämlich doppelt soviel Speicherplatz wie einfach genaue Zahlen; auch dauern Berechnungen mit diesen Zahlen länger. Weiterhin kann die

Fähigkeit zum Rechnen mit doppelt genauen Zahlen nicht schon vom Hersteller in Ihr FORTH-System eingebaut sein. Gegebenenfalls müssen Sie einige zusätzliche Blöcke in den Speicher laden, ehe Sie mit diesem Zahlentyp arbeiten können. In vielen FORTH-Implementationen gibt es zu den meisten arithmetischen Befehlen für einfach genaue Integers ein Gegenstück für doppelt genaue Zahlen. Diese wollen wir im vorliegenden Abschnitt besprechen. Da die meisten Details der Arbeitsweise für beide Operationstypen gleich sind, werden wir nicht allzu detailliert darauf eingehen.

D+ - Dies ist das Additionswort für doppelt genaue Integers; **D+** entfernt zwei doppelt genaue Integers vom Stack und legt dort ihre Summe (als doppelt genaue Integer) ab. Die Summanden, mit denen **D+** arbeitet, benötigen also vier Stack-Positionen. Leider gibt es keine Möglichkeit festzustellen, ob die Zahlen auf dem Stack einfach oder doppelt genau sind. Wenn Sie also vier einfach genaue Zahlen auf den Stack pushen und anschließend **D+** ausführen, dann werden die beiden obersten Stack-Positionen so behandelt, als handle es sich dabei um eine doppelt genaue Integer. Ebenso geht es mit dem dritten und vierten Stack-Eintrag. Obwohl sich also vier einfache genaue Integers auf dem Stack befinden, wird FORTH die doppelt genaue Addition mit ihnen ausführen, was natürlich das Ergebnis verfälscht. Die Stack-Relation für **D+** lautet:

$$d_1 \; d_2 \; \text{-->} \; d_{sum} \qquad (5\text{-}1)$$

Bekanntermaßen stellen wir mit d eine doppelt genaue Zahl dar, während n zur Darstellung von einfach genauen Integers dient. Anders gesagt: d steht für zwei Stack-Positionen, während n eine Stack-Position repräsentiert.

5.1.1 Ein- und Ausgabe von doppelt genauen Integers

Damit wir mit doppelt genauen Integers rechnen können, müssen wir sie irgendwie auf den Stack bringen. Glücklicherweise macht FORTH dies automatisch, wenn wir ihm anzeigen, daß es sich bei den Zahlen um doppelt genaue handelt. Dazu müssen wir nur irgendwo in der Zahl einen Dezimalpunkt mit aufnehmen. Wir könnten z.B. 34.56 schreiben. Dies behandelt FORTH als die doppelt genaue Integer

3456. Stören Sie sich nicht an dem Dezimalpunkt: Er dient nicht zur Kennzeichnung von Nachkommastellen, sondern stellt lediglich ein Signal für den FORTH-Interpreter dar. Genausogut hätten wir 3.456, 3456., 345.6 usw. schreiben können, denn all diese Darstellungen sind gleichwertig und bewirken, daß dieselbe Zahl (die doppelt genaue Integer 3456) auf den Stack gelegt wird. Über den Umgang mit Nachkommastellen (der sog. Gleitkommaarithmetik) werden wir im nächsten Abschnitt noch einiges erfahren. Im Kapitel 3-3 haben wir bereits erörtert, wie doppelt genaue Zahlen gespeichert werden. Erinnern Sie sich daran, daß die höchstwertigen Bits einer doppelt genauen Integer weiter oben im Stack abgelegt werden, so daß man eine positive einfach genaue Integer an oberster Stack-Position ganz einfach in eine doppelt genaue Zahl verwandeln kann, indem man eine 0 auf den Stack pusht (vgl. Abschnitt 3-3). Wenn wir also die Zahl 35 als doppelt genaue Integer speichern wollen, dann sind folgende Eingaben völlig gleichwertig:

```
35. (RETURN)                                        (5-2a)
35 0 (RETURN)                                       (5-2b)
```

Dieses Thema wird ausführlicher in Abschnitt 3-3 behandelt.

D. - Das FORTH-Wort **D.** entfernt die oberste doppelt genaue Integer vom Stack und gibt sie aus. **D.** funktioniert also genauso für doppelt genaue Integers, wie es das Punktkommando für einfach genaue Integers tut. Seine Stack-Relation:

```
d -->                                               (5-3)
```

Als Beispiel wollen wir ein FORTH-Wort schreiben, das die beiden obersten doppelt genauen Zahlen vom Stack entfernt und ihre Summe ausgibt:

```
: DOPPADD D+ D. ;                                   (5-4)
```

Wie Sie sehen können, greifen wir hier im wesentlichen auf dieselben Ideen wie bei der einfach genauen Arithmetik zurück.

D- - Das FORTH-Wort **D-** entfernt die beiden obersten doppelt genauen Integers vom Stack und legt dort ihre Differenz ab. Der oberste Stack-Eintrag wird dabei vom zweiten subtrahiert. Das Ergebnis ist ebenfalls doppelt genau. Wir haben folgende Stack-Relation:

$$d_1\ d_2 \text{ --> } d_{diff} \qquad (8\text{-}5)$$

Dabei ergibt sich d_{diff} als Ergebnis von d_1 minus d_2. Wieder sehen wir, daß **D-** im wesentlichen genauso funktioniert wie -, außer daß jetzt doppelt genaue Zahlen benutzt werden.

5.1.2 Multiplikation und Division

Die FORTH-Kommandos zur Multiplikation und Division von doppelt genauen Integers sind kein Bestandteil von FORTH-79. Sie sind jedoch im MMSFORTH und anderen FORTH-Systemen implementiert. Auch diese Wörter entsprechen ihren einfach genauen Gegenstücken. So entfernt z.B. **D*** die beiden obersten doppelt genauen Integers vom Stack und ersetzt sie durch ihr Produkt. Das Ergebnis ist ebenfalls eine doppelt genaue Integer.

Zur Division von doppelt genauen Zahlen benutzt man in FORTH die Wörter **D/** und **D/MOD**. Auch diese beiden Kommandos funktionieren ähnlich wie ihre einfach genauen Entsprechungen.

In vielen Programmen werden sowohl einfach genaue als auch doppelt genaue Integers benötigt. Die Größen, die ein Programm berechnet, können z.B. doppelt genau sein, während Schleifenparameter in der Regel einfach genaue Zahlen sind. Als Beispiel für das Mischen dieser beiden Zahlentypen schreiben wir unser Fakultätsprogramm erneut (vgl. Abb. 4-6), wobei wir diesmal doppelt genaue Integers verwenden. Beachten Sie, daß 8! = 40320 gilt. Diese Zahl kann nicht mehr als einfach genaue Integer dargestellt werden. Sie sehen das Programm in Abbildung 5-1. Gehen wir es einmal im Detail durch.

```
0 ( Doppelt genaue Fakultätsberechnung )
1 : FACT  1  +   1  0   1   4 ROLL   SWAP
2          DO   I 0   D*    LOOP  D.
3
4
```

ABBILDUNG 5-1: Ein FORTH-Wort, das die Fakultät mit doppelter Genauigkeit berechnet

Wir nehmen an, daß Sie mit dem Fakultätsprogramm aus Abbildung 4-6 bereits vertraut sind. Wenden wir uns jetzt Abbildung 5-1 zu. Das Programm geht davon aus, daß die Zahl, deren Fakultät berechnet werden soll, als einfach genaue Integer auf dem Stack liegt. Als erstes addieren wir 1 zu dieser Zahl. Im nächsten Schritt legen wir eine doppelt genaue 1 auf den Stack, und zwar, indem wir zuerst eine 1 und anschließend eine 0 auf den Stack pushen. Ebensogut hätten wir dem Interpreter anzeigen können, daß es sich um eine doppelt genaue Zahl handelt, indem wir 1. anstelle der 1 und der 0 eingeben. Nach Ausführung von 4 **ROLL** und **SWAP** enthält der Stack (von oben nach unten) eine 1, die Zahl, deren Fakultät berechnet werden soll erhöht um 1, und die doppelt genaue Integer 1. Mit dem FORTH-Wort **DO** auf Zeile 2 treten wir in die Schleife ein. Dadurch werden die beiden obersten Stack-Einträge vom Parameter-Stack entfernt und auf den Return-Stack gelegt. Innerhalb der Schleife besorgen wir uns als erstes den Schleifenindex durch das FORTH-Wort I und machen diesen zu einer doppelt genauen Zahl, indem wir eine Null auf den Stack pushen. Der restliche Teil des Programms funktioniert genauso, wie Sie es von der Abbildung 4-6 her schon kennen, außer daß die doppelt genaue Multiplikation benutzt wird. Da das Ergebnis doppelt genau ist, muß auch zur Ausgabe das Kommando **D.** geschrieben werden.

5.1.3 Stack-Manipulationen mit doppelt genauen Integers

Die dazu nötigen FORTH-Kommandos haben wir bereits ausführlich in Abschnitt 2-5 besprochen.

5.1.4 Vergleichswörter

Auch einige Vergleichswörter, die wir in Abschnitt 4-1 kennengelernt haben, haben ihre Entsprechung für doppelt genaue Zahlen. Das FORTH-Wort **D=** entspricht dem bekannten =. Das bedeutet, daß **D=** die beiden obersten doppelt genauen Integers vom Stack entfernt und durch ein Flag ersetzt. Dieses ist wahr, wenn die beiden Zahlen gleich waren. Ansonsten hat das Flag den Wahrheitswert "falsch". Denken Sie daran, daß ein Flag stets eine einfach genaue Integer ist. Deshalb haben wir folgende Stack-Relation:

$$d_1 \ d_2 \ \text{-->} \ n_{flag} \qquad (5\text{-}6)$$

Das FORTH-Wort **D<** entspricht <. Bei Ausführung von **D<** werden die beiden obersten doppelt genauen Integers vom Stack entfernt und durch ein Flag ersetzt. Dieses hat den Wahrheitswert "wahr", wenn die zweite doppelt genaue Integer auf dem Stack kleiner als die erste ist. Die Stack-Relation ist die gleiche wie in 5-6.

Für den Vergleich von doppelt genauen Zahlen stellt FORTH nicht so viele Wörter zur Verfügung wie für einfach genaue Zahlen. Diesen Mangel kann man jedoch einfach beheben, indem man sich seine eigenen Vergleichswörter schreibt. Wir können ein Wort für doppelt genaue Zahlen schreiben, das dem bekannten > entspricht; geben wir ihm den Namen **DD>**. Es soll die beiden obersten doppelt genauen Integers vom Stack entfernen und dort ein Flag ablegen. Dieses Flag wird "wahr", wenn die zweite doppelt genaue Integer auf dem Stack größer als die erste doppelt genaue Integer auf dem Stack ist. Die Definition dieses Wortes sehen Sie in Abbildung 5-2. Als erstes duplizieren wir die oberste doppelt genaue Integer. Als nächstes transportieren wir die ursprüngliche zweite doppelt genaue Integer an oberste Stack-Position, indem wir das Wort **2ROT** aufrufen. Diese Zahl wird als nächstes dupliziert. Ein weiteres **2ROT** sorgt dafür, daß der Stack nun folgendermaßen aussieht: d_2 d_1 d_1 d_2. Die ursprüngliche Stack-Konfiguration lautete: d_1 d_2. Jetzt führen wir den ersten Vergleich mit **D<** aus. Wir erhalten nun ein Flag auf dem Stack, das "wahr" ist, falls d_1 kleiner als ${}_2$ ist. Dieses Flag machen wir zu einer doppelt genauen Integer, indem wir zusätzlich eine 0 auf den Stack legen. Zweimalige Ausführung von **2ROT** bringt erneut die ursprünglichen zwei doppelt genauen Zahlen zuoberst auf den Stack. Da wir als nächstes einen Vergleich mit **D=** anstellen, spielt es keine Rolle, in welcher

Reihenfolge sich diese beiden Zahlen in den ersten beiden Stack-Positionen befinden. Der Vergleich mit **D=** hinterläßt ein Flag, unter dem sich auf dem Stack eine 0 und ein weiteres Flag befinden. **ROT** holt diese beiden Flags nun nach oben. Wenn eines von ihnen "wahr" ist, dann sollte **DD>** als Ergebnis ein "falsches" Flag liefern. Die einfache Addition mittels **+**, die in diesem Fall ausreichend ist, liefert uns nun entweder eine 0, eine 1 oder eine 2. Erhalten wir die 1 oder die 2, dann sollte **DD>** ein "falsches" Flag hinterlassen. Im Falle der 0 wird **DD>** aber wahr; diesen Fall testen wir durch **0=**. Wir erhalten somit nur dann ein "wahres" Flag, wenn beide zum Vergleich herangezogenen Flags auf dem Stack den Wahrheitswert "falsch" haben.

```
0 ( Doppelt genaues Vergleichswort )
1 : DD>  2DUP  2ROT  2DUP  2ROT
2        D<  0    2ROT  2ROT  D=  ROT
3        +   0=  .     DROP           ;
4
5
```

ABBILDUNG 5-2 Vergleich zweier doppelt genauer Integers

5.1.5 Weitere Befehle für doppelt genaue Integers

Wir stellen hier noch eine Reihe weiterer Befehle vor, die ähnlich wie einige bereits bekannte Wörter arbeiten, aber für doppelt genaue Integers vorgesehen sind.

DNEGATE - Das FORTH-Wort **DNEGATE** ist die doppelt genaue Entsprechung von **NEGATE** (vgl. Abschnitt 2-7). **DNEGATE** ändert das Vorzeichen der doppelt genauen Integer an oberster Stack-Position.

DMIN, DMAX und **DABS** - Diese drei FORTH-Wörter entsprechen den bereits bekannten Befehlen **MIN, MAX** und **ABS**, die Sie in Abschnitt 2-7 kennengelernt haben. **DMIN** entfernt z.B. die beiden obersten doppelt genauen Integers vom Stack und legt die kleinere der beiden dort ab. Das FORTH-Wort **DABS** ersetzt die oberste doppelt genaue Integer auf dem Stack durch ihren Absolutwert.

D#IN - Das FORTH-Wort **D#IN** entspricht dem bekannten **#IN** (vgl. Abschnitt 3-1). Das Kommando findet sich nicht in MMSFORTH. Bei Aufruf von **D#IN** unterbricht der FORTH-Interpreter seine Berechnungen und gibt ein Fragezeichen auf dem Bildschirm aus. Der Benutzer kann nun eine doppelt genaue Integer eingeben und anschließend die RETURN-Taste drücken. Selbst wenn die eingegebene Zahl keinen Dezimalpunkt enthält, so wird sie von **D#IN** als doppelt genaue Integer behandelt. Die Stack-Relation zu diesem Wort ist:

--> d (5-7)

5.2 Formatieren von Zahlen

Auch bei doppelt genauen Integers besteht die Möglichkeit zur formatierten Ausgabe, ähnlich, wie wir es bereits in Abschnitt 3-3 besprochen haben. Dabei muß man nur in einigen Fällen andere Befehle für doppelt genaue Integers verwenden, kann jedoch meistens auf die für einfache Zahlen bereits vertrauten Wörter zurückgreifen.

5.2.1 Datenfelder

Das FORTH-Wort **D.R** entspricht dem in Abschnitt 3-3 bereits diskutierten **.R.** Bei Ausführung von **D.R** sollten sich mindestens zwei Werte auf dem Stack befinden, nämlich eine einfach genaue Integer an oberster Stack-Position und darunter eine doppelt genaue Integer. **D.R** entfernt diese beiden Werte vom Stack. Die einfach genaue Integer gibt die Breite des Felds an, in dem die doppelt genaue Zahl ausgegeben werden soll. Wir haben also folgende Stack-Relation:

d n --> (5-8)

Ansonsten gelten alle in Abschnitt 3-3 angestellten Überlegungen für das Ausgeben von Zahlen in Datenfeldern auch für doppelt genaue Integers.

5.2.2 Zahlenausgabe mit Maske

Auch diese Art der Ausgabe haben wir bereits in Abschnitt 3-3 kennengelernt. Die Kenntnisse, die Sie dort erworben haben, können Sie ohne Änderung auf doppelt genaue Integers übertragen. Tatsächlich ist die Zahlenausgabe mit Maske ausschließlich für doppelt genaue Integers vorgesehen, weswegen wir in Kapitel 3-3 ja auch immer unsere einfach genauen Zahlen erst durch Hinzufügen einer Null zu doppelt genauen Zahlen machen mußten. Diesen Schritt können Sie sich bei der Ausgabe von doppelt genauen Integers mit Maske natürlich sparen. Ansonsten gilt das Kapitel 3-3 unverändert.

5.2.3 Ändern der Zahlenbasis

Auch hier gelten die Ausführungen von Kapitel 2-7 unverändert für doppelt genaue Integers. Wenn Sie so z.B. den Befehl **HEX** eingeben, dann wird jede Zahl, sei sie einfach oder doppelt genau, in hexadezimaler Schreibweise ausgegeben. (Natürlich werden auch Ihre Eingaben als Hexadezimalwerte interpretiert.)

5.3 Skalieren von Zahlen

Bei den bisherigen Rechenbeispielen mit Integers sind wir davon ausgegangen, daß die Ergebnisse unserer Programme stets korrekt sind. Dies ist jedoch nicht immer der Fall. Angenommen, wir wollen folgenden Ausdruck berechnen:

$$(2/4)*6 \qquad (5\text{-}9)$$

Der Bruch 2/4 entspricht 1/2 und 1/2 mal 6 ist gleich 3. Was aber macht FORTH aus diesem Ausdruck? Angenommen, wir geben folgendes ein:

```
6 2 4 / * (RETURN)                                    (5-10)
```

Der Befehl **/** dividiert 2/4. Dies liefert als Ergebnis jedoch 0 (vgl Abschnitt 2-1)! Wenn wir 0 mit 6 multiplizieren, dann erhalten wir wiederum 0. Daß wir hier ein falsches Ergebnis erhalten, ist kein Fehler von FORTH. Die Berechungen von FORTH sind genau. Der Fehler ergibt sich daraus, daß bei der Integer-Division der Divisionsrest verlorengeht. In diesem Fall haben wir es mit einem extremen Fall eines sog. Rundungsfehlers zu tun. Betrachten wir noch ein weiteres Beispiel. Der Benutzer gibt ein: 6 21 4 / * (RETURN). Dies sollte als Ergebnis (5,25) mal 6 ist gleich 31,5 liefern. FORTH berechnet jedoch 5*6=30. Auch hier geht bei der Integer-Division der Divisionsrest verloren. Die Division von 21/4 liefert deshalb das Ergebnis 5. Wir können die Genauigkeit der Berechnung (5-9) erhöhen, indem wir die Multiplikation **vor** der Division ausführen. Dazu schreiben wir (5-10) neu als

```
4 6 2 * SWAP / (RETURN)                               (5-11)
```

In diesem Fall wird zuerst das Produkt von 2 und 6 berechnet und liefert den Wert 12. Diesen dividieren wir anschließend durch 4, wodurch wir das Ergebnis 3 erhalten. Jetzt haben wir die richtige Antwort. Nun zu dem zweiten Beispiel; hier sieht der Stack so aus: 4 6 21. Wir multiplizieren zuerst und erhalten 21*6=126. Die Division von 126/4 sollte 31,5 ergeben, bekanntermaßen wird bei Integer-Division jedoch der Rest fallen gelassen, so daß sich das Ergebnis 31 einstellt. Bei unserem vorherigen Berechnungsversuch erhielten wir 30. Natürlich ist das Ergebnis 31 genauer als das Ergebnis 30.

*Diese Beispiele haben gezeigt, daß man die Genauigkeit von Re*chenergebnissen verbessern kann, indem man die Multiplikationen in einem Ausdruck möglichst **vor** den Divisionen ausführt. Dies ist jedoch nicht immer möglich. Selbst wenn wir nur mit Zahlen arbeiten, die verhältnismäßig klein sind, so können die als Zwischenergebnis anfallenden Produkte trotzdem für das System zu groß werden. Bei der Arbeit mit einfach genauen Integers müssen ja alle Zahlen zwischen -32768 und 32767 liegen. Nehmen Sie jetzt einmal an, daß wir den Ausdruck (459)*(734/915) zu berechnen haben. Das Produkt 363 906 ist zu groß zur Darstellung in einer einfach genauen Integer, weswegen die Berechnung in einem fehlerhaften Ergebnis endet. (Oftmals zeigt Ihr System diesen Fehler

gar nicht erst an!) Andererseits ist das Endergebnis 368,2 nicht zu groß für das System. Wir könnten diese Schwierigkeit umgehen, indem wir mit doppelt genauen Zahlen arbeiten. Diese benötigen aber bekanntermaßen mehr Rechenzeit und Speicherplatz. Zum Glück stellt FORTH ein Wort bereit, das die Arbeit mit großen Produkten als Zwischenergebnis erlaubt, ohne daß wir bei unseren Berechnungen auf doppelt genaue Integers zurückgreifen müssen.

*/ - Das FORTH-Wort ***/** arbeitet mit drei einfach genauen Integers auf dem Stack. Es hat folgende Stack-Relation:

$$n_1\ n_2\ n_3\ \text{-->}\ n_{erg} \qquad (5\text{-}12)$$

Seine Arbeitsweise ist wie folgt: Zuerst wird das Produkt $n_1 * n_2$ berechnet. Das Ergebnis dieser Multiplikation wird als doppelt genaue Integer gespeichert. Diese wird anschließend durch n_3 dividiert. Das Endergebnis wird als einfach genaue Integer auf den Stack gepusht. Dadurch, daß das Zwischenergebnis (das Produkt) doppelt genau gespeichert ist, ergeben sich keine Überlaufprobleme, (d.h., es kann nicht passieren, daß das Zwischenergebnis zu groß wird). Der Benutzer muß sich jedoch bei der Arbeit in seinem Programm nicht mit den Details für den Umgang mit doppelt genauen Integers befassen. Wir können jetzt die Berechnungen aus Beispiel 5-10 neu schreiben:

```
6 2 4 */ (RETURN)                                        (5-13)
```

***/MOD** - Das FORTH-Wort ***/MOD** funktioniert genauso wie ***/**, außer daß es zwei einfach genaue Integers als Ergebnis liefert. Bei der einen handelt es sich um den Quotienten, während die andere der Divisionsrest ist. Wir haben also folgende Stack-Relation:

$$n_1\ n_2\ n_3\ \text{-->}\ n_{rest}\ n_{quot} \qquad (5\text{-}14)$$

Der oberste Stack-Eintrag nach Ausführung von ***/MOD** stellt den Quotienten der Division dar, während der zweite Stack-Eintrag der Divisionsrest ist.

D*/ und **D*/MOD** - Diese beiden FORTH-Wörter funktionieren genauso wie ***/** und ***/MOD**, außer daß sie mit doppelt genauen Zahlen arbeiten. Als Zwischenergebnis wird ein Produkt in zwei aufeinanderfolgenden doppelt genauen Integers abgelegt, d.h., das Zwischenergebnis wird in vierfacher Genauigkeit dargestellt. Dadurch können wir die Vorteile der beiden eben eingeführten Arithmetik-Wörter auch für doppelt genaue Integers nutzen. **D*/** und **D*/MOD** sind ebenfalls nicht Teil von FORTH-79, jedoch im MMSFORTH implementiert.

5.3.1 Skalieren von Berechnungen

Was macht man, wenn man mit Zahlen mit Nachkommastellen (z.B. Geldbeträgen) rechnen muß, jedoch nur Integers in seinem System zur Verfügung hat? In diesem Fall bedient man sich eines Verfahrens, das den Namen Skalierung trägt. Als Beispiel nehmen wir uns vor, die Provision eines Verkäufers zu berechnen. Dabei müssen wir mit DM- und Pfennigbeträgen rechnen. Außerdem ist es erforderlich, einen bestimmten Prozentsatz der Umsätze des Verkäufers zu berechnen. All diese Operationen haben mit Dezimalbrüchen zu tun. Dennoch können wir in unseren Berechnungen mit Integers arbeiten. Angenommen, wir haben es mit einem Betrag von 56,42 DM zu tun. Wenn wir diesen Betrag mit 100 multiplizieren, dann können wir ihn als die Integer 5642 darstellen. Wenn DM-Beträge mit 100 multipliziert werden, dann kann man sie also als Integers repräsentieren. In diesem Fall sagt man, daß der Betrag mit dem Skalierungsfaktor 100 multipliziert wurde. Auch bei den eben eingeführten Worten ***/** und ***/MOD** kann man von einer bestimmten Art von Skalierung sprechen. Schließlich skaliert FORTH automatisch doppelt genaue Integers; wenn Sie eine Zahl mit einem Dezimalpunkt an beliebiger Stelle innerhalb der Ziffernfolge eingeben, dann wird diese Zahl automatisch als doppelt genaue Integer behandelt. Wenn Sie aber nicht mit automatischer Skalierung arbeiten, sondern die Skalierungsfaktoren selbst bestimmen, dann müssen Sie sehr vorsichtig vorgehen. Insbesondere müssen alle Werte mit demselben Faktor skaliert werden. Bei Eingabe von 45.78 und 4.578 behandelt FORTH diese beiden Werte als die gleiche Zahl. Sieht man diese beiden Werte jedoch als Geldbeträge (mit Dezimalpunkt, wie in Amerika üblich) an, dann ist der eine Betrag zehnmal größer als der andere.

Jetzt aber zurück zu unserem Beispiel, der Berechnung von Verkäuferprovisionen. Wir schreiben ein FORTH-Wort, das die Beträge der einzelnen Verkäufe addiert und dann 15% der sich daraus ergebenden Summe berechnet. Wir werden für dieses Problem zwei Lösungen liefern, wobei die eine mit einfach genauen, die andere mit doppelt genauen Integers arbeitet. Beide Lösungen finden Sie in Abbildung 5-3.

```
 0 (Beispiel fuer Skalierung)
 1 : SUMME      1   DO  +    LOOP   ;
 2 : PROVISION   SUMME  15  100        */
 3          CR   0   <#  #  #  44 HOLD   #S   #>   TYPE ;
 4
 5
 6 (Zweite Fassung mit doppelt genauen Integers)
 7 : DSUMME      1  DO    D+   LOOP   ;
 8 : DPROVISION   DSUMME  15.  100.      D*/
 9          CR     <#  #  #   44 HOLD   #S   #>   TYPE ;
10
11
12
13
14
15
```

ABBILDUNG 5-3: Ein Beispiel für die Skalierungstechnik

Untersuchen wir als erstes das Programm, das mit einfach genauen Zahlen arbeitet. Das Hauptwort trägt den Namen **PROVISION** und ruft seinerseits das Unterwort **SUMME**. Nun zuerst zu **SUMME**. Dieses berechnet die Summe der einzelnen Verkäufe. Dazu wird der Betrag jedes Verkaufs auf den Stack gepusht und zuletzt die Anzahl der einzelnen Verkäufe. Die Zahlen, die den Verkaufswert darstellen, sollten als Pfennigbeträge eingegeben werden. Wenn der Vertreter also drei Verkäufe mit einem Wert von 45,09 DM, 56,32 DM und 103,45 DM getätigt hat, dann müssen Sie folgendes eingeben:

4509 5632 10345 3 (5-15)

Wie Sie sehen können, sind die solcherart eingegebenen Daten bereits skaliert. **SUMME** legt nun, wenn es gerufen wird, eine 1 auf den Stack und tritt in eine **DO**-Schleife ein. Das **DO**-Wort findet vorschriftsmäßig den Anfangswert für den Schleifenindex und den Testwert auf dem Daten-Stack vor, welche es von dort entfernt und auf den Return-Stack legt. Im Falle des Beispiels 5-15 bedeutet dies, daß zwei Schleifendurchgänge stattfinden, d.h., in diesem Fall, daß zweimal addiert wird. Nach Verlassen der Schleife haben wir also die Summe der vom Verkäufer getätigten Umsätze. Wenden wir uns nun dem Hauptwort **PROVISION** zu. Dieses ruft als erstes das bereits besprochene **SUMME**, um den Gesamtumsatz zu erhalten. Um nun 15 % von dieser Summe berechnen zu können, pushen wir 15 und 100 (in dieser Reihenfolge) auf den Stack und rufen das arithmetische Wort ***/**. Dies bewirkt, daß der Gesamtumsatz mit 15 multipliziert und anschließend durch 100 dividiert wird. Wir haben somit die gewünschten 15 % des Gesamtumsatzes berechnet. Diesen Betrag wollen wir jetzt ausdrucken, wobei sich das Dezimalkomma an der richtigen Stelle befindet. Dazu dienen die Anweisungen in Zeile 3, die ein weiteres Beispiel für die Zahlenausgabe mit Maske (vgl. Abschnitt 3-3) darstellen. Die auszugebende Zahl wird erst einmal doppelt genau gemacht, indem eine 0 auf den Stack gepusht wird. Die Ausgabemaske bewirkt, daß die beiden niedrigstwertigen Dezimalziffern der Zahl rechts vom Dezimalkomma ausgegeben werden. Dies ist völlig analog zu den Beispielen in Abschnitt 3-3.

Wenden wir uns nun dem zweiten Beispiel von Abbildung 5-3 zu, welches mit doppelt genauen Integers arbeitet. Jetzt sollten Sie bei der Dateneingabe den (in Amerika üblichen) Dezimalpunkt mit eingeben und dabei das übliche Format für Geldbeträge verwenden. Die Daten aus Beispiel 5-15 müssen also folgendermaßen eingegeben werden:

```
45.09 56.32 103.45 3                                   (5-16)
```

Die Anzahl der Eingabedaten (3) geben wir nach wie vor als einfach genaue Integer an. Das Wort zur Summenbildung - **DSUMME** - funktioniert im wesentlichen genauso wie **SUMME**, außer daß doppelt genaue Addition mittels **D+** ausgeführt wird. Ebenso ist **DPROVISION** im wesentlichen dem Wort **PROVISION** ähnlich. Unterschiede ergeben sich daraus, daß wir zur Berechnung des Prozentsatzes den doppelt genauen Operator **D*/** verwenden wollen. Deshalb müssen wir die

Konstanten in der Form 15. und 100. eingeben, damit sie von FORTH als doppelt genaue Werte interpretiert werden. Weil **D*/** ein doppelt genaues Ergebnis liefert, müssen wir nicht erst eine Null auf den Stack hieven, ehe wir das Ergebnis mit der Druckmaske ausgeben können.

Nicht alle FORTH-Systeme verfügen über **D*/**. Wenn Ihr System allerdings die Wörter **D*** und **D/** kennt, dann können Sie trotzdem mit **DPROVISION** arbeiten; allerdings ist es dann nicht möglich, Zwischenergebnisse mit vierfacher Genauigkeit zu speichern.

5.4 Berechnungen im gemischten Modus

Manchmal ist es nötig, bei einer Berechnung sowohl mit einfachen als auch mit doppelt genauen Integers zu arbeiten. Wir wissen ja bereits, daß manchmal das Produkt zweier einfach genauer Integers zu groß ist, um in einer einfach genauen Integer Platz zu finden. Wenn beide Zahlen positiv sind, dann können wir mit der bereits bekannten Methode (0 auf den Stack pushen) zur rechten Zeit aus den Multiplikanden doppelt genaue Zahlen machen und unser Ergebnis mit doppelt genauer Multiplikation berechnen. Einige FORTH-Systeme verfügen jedoch über Wörter, die dem Benutzer in diesen Situationen die Arbeit erleichtern. Sie sind zwar kein Teil von FORTH-79, stehen jedoch in MMSFORTH und anderen FORTH-Systemen zur Verfügung. Kommen in einer Berechnung unterschiedliche Zahlentypen - in diesem Fall einfache und doppelt genaue Integers - vor, so spricht man von Berechnungen im gemischten Modus. Dem gemischten Modus gilt unser Interesse in diesem Abschnitt.

M* - Das FORTH-Wort **M*** entfernt zwei einfach genaue Integers vom Stack, berechnet ihr Produkt als doppelt genaue Integer und legt diese auf den Stack. Wir erhalten folgende Stack-Relation:

$$n_1\ n_2\ \text{-->}\ d_{prod} \qquad (5\text{-}17)$$

So berechnet beispielsweise das folgende Wort das Produkt zweier einfach genauer Zahlen mit doppelter Genauigkeit und gibt es auch als doppelt genaue Integer aus:

```
: MIXMULT M* D. ;                                    (5-18)
```

M/ und **M/MOD** - Wir wissen bereits, daß es oftmals sehr gelegen kommt, das Produkt zweier einfach genauer Zahlen mit doppelter Genauigkeit zu erhalten. Umgekehrt kann es auch nützlich sein, sowohl den Quotienten als auch den Rest der Division einer doppelt genauen Integer durch eine einfach genaue Integer mit einfacher Genauigkeit darzustellen. Bei einer solchen Operation im gemischten Modus liefert das Wort **M/** den Quotienten mit einfacher Genauigkeit. Wir erhalten folgende Stack-Relation:

$$d\ n \text{ --> } n_{quot} \qquad (5\text{-}19)$$

Beachten Sie, daß die doppelt genaue Integer (an zweiter und dritter Stack-Position) durch die einfach genaue Integer an oberster Stack-Position dividiert wird. Diese beiden Zahlen (also insgesamt drei Stack-Positionen) werden vom Stack entfernt und durch den einfach genauen Quotienten ersetzt.

Das Wort **M/MOD** funktioniert ähnlich wie **M/**, außer daß es als Ergebnis zwei einfach genaue Integers liefert. Eine davon ist der Quotient, während die andere den Divisionsrest darstellt. Die Stack-Relation lautet:

$$d\ n \text{ --> } n_{rest}\ n_{quot} \qquad (5\text{-}20)$$

Beachten Sie, daß sich der Quotient an oberster Stack-Position befindet.

M+ und **M-** - Zum Addieren einer doppelt genauen Integer mit einer einfach genauen Integer dient das Wort **M+**. Das Ergebnis dieser Operation ist natürlich eine doppelt genaue Integer, weswegen die Stack-Relation lautet:

$$d\ n \text{ --> } d_{sum} \qquad (5\text{-}21)$$

Beachten Sie, daß sich die einfach genaue Zahl an oberster Stack-Position befinden muß. Auch Subtraktionen im gemischten Modus sind möglich; dazu bedient man sich des Wortes **M-**. Es hat die Stack-Relation:

$$d\ n \dashrightarrow d_{dif} \qquad (5\text{-}22)$$

Wie Sie sehen, wird die einfach genaue Integer an oberster Stack-Position von der doppelt genauen Integer subtrahiert, die die zweite und dritte Stack-Position belegt. Das Ergebnis, welches die beiden Argumente der Operation ersetzt, wird in doppelter Genauigkeit dargestellt.

Sollte Ihr FORTH-System über **M-** nicht verfügen, so können Sie sich zumindest für positive Zahlen leicht ein eigenes Wort definieren, das dieselbe Wirkung hat. Seine Definition lautet:

```
: MIX- 0 D- ;                                    (5-23)
```

Mit dem erprobten Trick machen wir aus der einfach genauen Integer an oberster Stack-Position eine doppelt genaue, so daß wir das Wort **D-** rufen können.

M*/ - Dieses Wort funktioniert ähnlich wie ***/** oder **D*/**, außer daß es im gemischten Modus arbeitet, also das Produkt einer doppelt genauen Integer mit einer einfach genauen Integer berechnet, dieses Ergebnis mit dreifacher Genauigkeit speichert (also drei Stack-Positionen dazu benötigt) und anschließend durch eine einfach genaue Integer dividiert. Der Quotient dieser Division wird mit einfacher Genauigkeit dargestellt. Es ergibt sich folgende Stack-Relation:

$$d\ n_1\ n_2 \dashrightarrow n_{quot} \qquad (5\text{-}24)$$

Dabei wird das Produkt aus d und n_1 berechnet und anschließend durch n_2 dividiert.

5.5 Vorzeichenlose Zahlen

Alle in den bisherigen Programmbeispielen verwendeten Zahlen waren mit einem Vorzeichen versehen. Sie konnten also entweder positiv oder negativ sein, was bei der Ausgabe durch ein vorgestelltes Minuszeichen bzw. das Fehlen desselben angezeigt wird. Sie haben bereits erfahren, daß Ihr Computer Zahlen intern als Folgen von Nullen und Einsen (sog. Bits) speichert, also in der Binärdarstellung (vgl. Abschnitt 1-2). Eines dieser Bits benutzt der Rechner dabei zur Speicherung des Vorzeichens. Allerdings ergibt sich eine negative Zahl nicht einfach aus einer positiven, indem man das entsprechende Bit verändert! Arbeiten wir nun ausschließlich mit positiven Zahlen, dann brauchen wir keine Vorkehrungen für negative Zahlen zu treffen und können deshalb auf dieses Vorzeichenbit verzichten. Es steht dann zur Verfügung, um den Absolutbetrag der Zahl zu repräsentieren, wodurch der Wertevorrat der darstellbaren Zahlen vergrößert wird. In einem typischen FORTH-System werden in der Regel 16 Bits für die Darstellung von vorzeichenbehafteten einfach genauen Integers verwendet. Wie bereits mehrfach ausgeführt, können solche Zahlen im Bereich zwischen -32768 und 32767 liegen. Wenn wir nun diese 16 Bits zur ausschließlichen Darstellung von positiven Zahlen benutzen, dann können wir damit Zahlen zwischen 0 und 65535 repräsentieren. FORTH erlaubt es, einfach genaue Integers so zu behandeln, als ob sie kein Vorzeichen hätten. Allerdings sollten Sie sich darüber im klaren sein, daß Sie einer Zahl auf dem Stack nicht ohne weiteres ansehen können, ob sie vorzeichenlos oder mit einem Vorzeichen versehen ist. Es liegt allein in Ihrer Verantwortung, ob das Programm die Zahlen richtig interpretiert.

U. - Das FORTH-Kommando **U.** arbeitet wie das Punktkommando in FORTH, außer daß es davon ausgeht, daß die auszugebende Zahl vorzeichenlos ist. Seine Stack-Relation lautet:

```
    u -->                                               (5-25)
```

Wie Sie sehen können, stellen wir vorzeichenlose Zahlen auf dem Stack durch den Buchstaben "u" dar. Der Platzbedarf von vorzeichenlosen Zahlen ist der gleiche wie der von einfach genauen Integers; Zahlen vom Typ "u" und "n" beanspruchen also beide genau eine Stack-Position. Man kann also sagen, daß **U.** die ober-

ste einfach genaue Integer vom Stack entfernt und als vorzeichenlose Zahl ausgibt. Wir wollen einmal an einem kleinen Beispielprogramm den Unterschied zwischen vorzeichenlosen Zahlen und solchen mit Vorzeichen demonstrieren. Betrachten Sie das folgende Wort:

```
: CHECK DUP U. . ;                                    (5-26)
```

Offensichtlich dupliziert **CHECK** die oberste einfach genaue Zahl auf dem Stack und gibt sie einmal als vorzeichenlose und einmal als "gewöhnliche" Integer aus. Bei Eingabe von

```
3 CHECK
```

erhalten wir als Ergebnis

```
3 3
```

Wenn eine Integer also positiv ist, dann ist ihre vorzeichenlose Interpretation identisch mit der vorzeichenbehafteten. Ganz anders sieht es bei negativen Zahlen aus. Probieren Sie einmal das folgende:

```
-1 CHECK
```

Dies liefert das Ergebnis

```
65535 -1
```

Dies bedeutet, daß die Binärdarstellung der vorzeichenbehafteten Zahl -1 gleich ist wie die der vorzeichenlosen 65535.

U.R - Das FORTH-Wort **U.R** dient der Ausgabe von vorzeichenlosen Zahlen und entspricht weitgehend dem bereits bekannten **.R** (vgl. Abschnitt 3-3). **U.R** erwartet jedoch, daß es eine vorzeichenlose Zahl ausgeben soll. Seine Stack-Relation lautet:

```
u n -->                                               (5-27)
```

Wie bei **.R** gibt der oberste Stack-Eintrag die Feldbreite an, während der zweite Stack-Eintrag die auszugebende Zahl darstellt. Beide Werte werden durch **U.R** vom Stack entfernt. Ist das angegebene Feld nicht breit genug für die Ausgabe dieser Zahl, so kümmert sich FORTH nicht um diese Angabe und legt selbst ein ausreichend breites Feld an.

U* - Dieses FORTH-Kommando dient der Multiplikation zweier vorzeichenloser Integer. **U*** entfernt die obersten zwei Zahlen vom Stack, bildet ihr Produkt und legt es als vorzeichenlose doppelt genaue Integer auf den Stack. Dadurch wird der Darstellungsbereich des Ergebnisses noch einmal beträchtlich vergrößert, da vorzeichenlose doppelt genaue Integers bis zu 4294967295 groß sein können. Wir haben folgende Stack-Relation:

$$u_1 \; u_2 \; \text{-->} \; u_d \qquad (5\text{-}28)$$

Zur Darstellung einer doppelt genauen vorzeichenlosen Integer verwenden wir also ein "u" mit dem Index "d". Zur Ausgabe dieses Zahlentyps gibt es keine eigenen FORTH-Worte; dies kommt daher, weil dieser Datentyp hauptsächlich zur Speicherung von Zwischenergebnissen dient. Wir wenden uns deshalb der Frage zu, wie man aus vorzeichenlosen doppelt genauen Integers vorzeichenlose einfach genaue Integers machen kann.

U/MOD - Das FORTH-Kommando **U/MOD** sorgt dafür, daß die vorzeichenlose doppelt genaue Integer in zweiter und dritter Stack-Position durch die vorzeichenlose einfach genaue Integer an erster Stack-Position dividiert wird. Diese beiden Werte werden vom Stack entfernt, und an ihrer Stelle werden Quotient und Divisionsrest dort abgelegt. Es ergibt sich folgende Stack-Relation:

$$u_d \; u \; \text{-->} \; u_{rest} \; u_{quot} \qquad (2\text{-}29)$$

Sowohl Quotient als auch Rest werden als vorzeichenlose einfach genaue Zahlen ausgegeben. Denken Sie daran, daß Sie sich diese Werte mit dem Kommando **U.** und nicht einfach mit dem Punktkommando ansehen müssen, um das korrekte Ergebnis zu erfahren.

U< - Dieses FORTH-Wort dient zum Vergleich zweier vorzeichenloser Zahlen. Es liefert ein Flag, das "wahr" ist, wenn die zweite Zahl auf dem Stack kleiner ist als die erste Zahl auf dem Stack. Beide zu vergleichenden Zahlen müssen vorzeichenlose Integers sein, weswegen sich folgende Stack-Relation ergibt:

$$u_1 \ u_2 \ \text{-->} \ n_{flag} \tag{5-30}$$

Das Flag ist also nur dann "wahr", wenn u_1 kleiner als u_2 ist.

DO und **/LOOP** - Um die Anzahl der Durchgänge durch eine unbedingte Schleife größer zu machen, ist es möglich, diese Parameter als vorzeichenlose Integers anzugeben. Solche Schleifen müssen allerdings mit Hilfe des Wortpaares **DO** und **/LOOP** aufgebaut werden. Die beiden Wörter funktionieren genau wie die bereits bekannten **DO** und **+LOOP**, außer daß die Schleifenparameter als vorzeichenlose einfach genaue Zahlen behandelt werden. Ansonsten gilt für solche Schleifen alles, was bereits in Abschnitt 4-3 gesagt wurde.

5.6 Bit-Operationen

Wie Sie wissen, speichert Ihr Computer intern alle Zahlen in der Binärdarstellung. Die Binärdarstellung einer Zahl besteht aus einer Folge von Nullen und Einsen. Jede dieser Nullen bzw. Einsen trägt den Namen Bit. Gelegentlich ist es nützlich, die einzelnen Bits, die zur Darstellung einer Binärzahl dienen, gezielt manipulieren zu können. In diesem Abschnitt werden Sie FORTH-Wörter kennenlernen, mit denen das möglich ist.

Wenn einzelne Bits manipuliert werden, dann geschieht dies mit Hilfe sog. logischer Operatoren. Einige dieser Operatoren werden sehr häufig gebraucht. Es sind dies das logische UND, das logische ODER und das logische exklusive Oder (XOR). Diese Operatoren vergleichen stets zwei Bitwerte miteinander und liefern in Abhängigkeit davon einen neuen Bitwert. Das logische UND (engl. AND) vergleicht zwei Bits; sind sie beide auf 1 gesetzt, dann liefert auch UND den Wert 1. Ist nur eines der beiden Bits (oder auch beide) auf Null gesetzt, so hat auch die UND-Operation den Wert 0. Auch das logische ODER vergleicht zwei Bits. Wenn nur eines

der beiden (oder auch beide) Bits auf 1 gesetzt ist, dann hat ODER den Wert 1. Die Operation des logischen ODER liefert also nur dann den Wert Null, wenn beide Vergleichsbits den Wert Null haben. Die dritte logische Operation, die häufig anzutreffen ist, das exklusive Oder (Fachausdruck: XOR) verhält sich ähnlich wie die ODER-Operation. Wenn eines der beiden Vergleichsbits den Wert 1 hat, dann wird auch XOR den Wert 1 liefern. Zusätzlich hat XOR aber auch noch den Wert 0, wenn beide Vergleichsbits den Wert 1 haben. Anders ausgedrückt kann man sagen, daß XOR nur dann den Wert 1 liefert, wenn die beiden Vergleichsbits verschiedene Werte haben; sind sie beide gleich 0 oder gleich 1, dann ergibt XOR auch den Wert 0. Die FORTH-Wörter zur Ausführung dieser logischen Operationen sind kein Teil des Standard-FORTH. Sie finden sich jedoch im MMSFORTH und in anderen FORTH-Systemen.

AND - Das FORTH-Wort **AND** entfernt die beiden obersten einfach genauen Integers vom Stack und vergleicht sie bitweise unter Verwendung der oben beschriebenen UND-Operation. Dabei wird das erste Bit der ersten Zahl mit dem ersten Bit der zweiten Zahl über logisches UND verglichen. Sind beide Bits gleich 1, dann wird auch das erste Bit in der Ergebniszahl den Wert 1 haben; ist nur eines der Vergleichsbits 0, dann hat auch das erste Bit in der Ergebniszahl den Wert 0. Als nächstes wird das zweite Bit der ersten Zahl mit dem zweiten Bit der zweiten Zahl verglichen und der Prozeß wiederholt. Jedes der 16 Bits einer einfach genauen Integer wird somit individuell dem Vergleich mit der logischen UND-Operation unterworfen. Dieser Prozeß hat eine neue 16-Bit-Zahl zum Ergebnis, welche von AND auf den Stack gelegt wird. Deswegen lautet die Stack-Relation:

$$n_1\ n_2 \dashrightarrow n \qquad (5\text{-}31)$$

Wir wollen ein Beispiel für den Einsatz von **AND** geben. Um der Klarheit willen schreiben wir die Zahlen in Binärdarstellung. Da die meisten FORTH-Systeme die Zahlenbasis wechseln können, ist dies durchaus zulässig: Der Programmierer braucht nur zu vereinbaren, daß er jetzt in der Binärdarstellung zu arbeiten wünscht (vergleichen Sie dazu Abschnitt 2-7). Unter der Voraussetzung, daß wir im Binärsystem (Basis 2) arbeiten, wollen wir jetzt das folgende FORTH-Wort definieren.

```
: ANDTEST AND . ;                                    (5-32)
```

Jetzt tippen wir:

```
10011001 01110001 ANDTEST (RETURN)
```

als Ergebnis erhalten wir:

```
00010001
```

Lediglich die erste und fünfte Bit-Position (von rechts gezählt) ist in beiden Vergleichszahlen auf 1 gesetzt. Deshalb erhalten wir als Ergebnis des FORTH-Wortes **AND** eine Zahl, bei der nur die erste und die fünfte Bit-Position auf 1 gesetzt ist. (Möglicherweise gibt Ihr FORTH-System führende Nullen bei der Darstellung einer Binärzahl nicht aus.) Hätten wir bei der Zahleneingabe im Dezimalsystem gearbeitet, dann würde das zum selben Ergebnis geführt haben, außer daß die Zahlen natürlich in ihrer dezimalen Schreibweise dargestellt worden wären.

OR - Das FORTH-Wort **OR** funktioniert ähnlich wie das bereits bekannte **AND**, außer daß der bitweise Vergleich der beiden Zahlen auf dem Stack mit der logischen ODER-Operation erfolgt. Die Stack-Relation geht ebenfalls aus (5-31) hervor. Hier ein Beispiel:

```
: ORTEST OR . ;                                      (5-33)
```

Wenn wir jetzt eingeben:

```
10011001 01110001 ORTEST (RETURN)
```

dann erhalten wir:

```
1111101
```

XOR - Das FORTH-Kommando **XOR** funktioniert im wesentlichen genauso wie **OR**, außer daß für die Bit-Vergleiche die Operation des exklusiven Oder herangezogen wird.

5.7 Gleitkommaarithmetik

Bisher konnten wir nur mit ganzen Zahlen rechnen. Wenn Geldbeträge an unseren Berechnungen beteiligt waren, haben wir uns mit Skalierung beholfen, um aus diesen ganzzahlige Summen zu bilden. Bei der Arbeit mit Programmiersprachen gibt es jedoch einen grundlegenden Unterschied zwischen Operationen mit ganzen Zahlen (Integers) und Operationen mit Gleitkommazahlen. Ein Computer benötigt zur Darstellung einer Gleitkommazahl in der Regel zwei Teile: die Mantisse und den Exponenten. Bei der Ausgabe einer Zahl wird der Exponent meist als Zehnerpotenz geschrieben. Diese Darstellung von Gleitkommazahlen - die sog. wissenschaftliche Notation - schreibt die Zahl 56,7 in der Form $.567*10^2$. Dabei ist die Zahl .567 die Mantisse, während es sich bei 10^2 um den Exponenten der Gleitkommazahl handelt. Wir schreiben also:

$$.567 \times 10^2 = 56.7 \qquad (5\text{-}34a)$$

Viele Computer zeigen die 10er-Potenz des Exponenten durch den Buchstaben E an, d.h.:

$$56.7 = .567 \text{ E } 2 \qquad (5\text{-}34b)$$

Betrachten wir noch einige weitere Gleitkommazahlen:

E2=100
E6=1000000
E-2=.01
E-4=.0001

Durch den Einsatz von Gleitkommazahlen können wir sowohl sehr große als auch sehr kleine Zahlen darstellen und sind außerdem in der Lage, mit Dezimalbrüchen zu arbeiten. Bei kleineren Computern reicht der Exponent in der Regel von 10^{-38} bis 10^{+37}. Damit läßt sich eine enorme Menge an Zahlen darstellen. Es gibt jedoch einen Nachteil bei den Gleitkommazahlen: Das Arbeiten mit ihnen - die sog. Gleitkommaarithmetik - ist wesentlich speicherplatz- und rechenzeitaufwendiger als die einfachere Integer-Arithmetik. Dies kommt daher, daß der Computer beim Arbeiten mit Gleitkommazahlen

sowohl über den Exponenten als auch über die Mantisse Buch führen muß. Ein weiterer Nachteil: Integer-Operationen sind - solange sich kein Werteüber- oder Unterlauf ergibt - stets exakt. Bei Gleitkommaoperationen ergeben sich jedoch bei jedem Rechenschritt Rundungsfehler. Sie sollten also in Ihren FORTH-Programmen so oft wie möglich Integerarithmetik einsetzen. Manchmal ist es jedoch unumgänglich, mit Gleitkommazahlen zu rechnen. In FORTH-79 ist dieser Zahlentyp gar nicht vorgesehen. Im MMSFORTH und anderen FORTH-Systemen stehen Gleitkommazahlen jedoch zur Verfügung, weswegen wir sie hier darstellen. Für die Arbeit mit Gleitkommazahlen gibt es in FORTH keinen Standard. Deshalb können auch die Worte, die mit diesem Zahlentyp operieren, auf jedem System anders lauten. Die grundlegenden Ideen für die Verarbeitung von Gleitkommazahlen sind jedoch auf allen Systemen gleich. Deshalb werden wir uns im Folgenden auch mehr den elementaren Fragen widmen und nicht so sehr auf Details eingehen.

5.7.1 Gleitkommaarithmetik in FORTH

Wenn wir eine Gleitkommazahl eingeben wollen, dann müssen wir dies dem Computer irgendwie anzeigen. Dazu stellen wir der Zahl ein Prozentzeichen, gefolgt von einem Leerzeichen, voran. Mittels

```
% 6 % 7 (RETURN)                                    (5-35)
```

legen wir also 6 und 7 als Gleikommazahlen auf den Stack. Auch hier gilt, daß man einer Zahl auf dem Stack nicht ansehen kann, ob es sich dabei um eine Gleikommazahl handelt, eine einfach genaue oder eine doppelt genaue Integer. Gleitkommazahlen benötigen zu ihrer Speicherung ebenso wie doppelt genaue Integers zwei Stack-Positionen.

F#IN - Dieses Wort wendet man an, um den Benutzer zur Eingabe einer Gleitkommazahl aufzufordern. Es funktioniert genauso wie das bereits bekannte **#IN**, außer daß es eine Gleitkommazahl auf dem Stack hinterläßt.

F. und **F.R** - Zur Ausgabe einer Gleitkommazahl setzen wir das FORTH-Wort **F.** ein. Dies entfernt die oberste Gleitkommazahl vom Stack und gibt sie auf dem Bildschirm aus. Die Stack-Relation lautet:

f --> (5-36)

Wie Sie sehen, benutzen wir den Buchstaben "f", um Gleitkommazahlen (engl. "floating point numbers") auf dem Stack anzuzeigen. Denken Sie daran, daß jedes "f" ebenso wie jedes "d" zwei Stack-Positionen beansprucht.

Auch Gleitkommazahlen können in Datenfeldern ausgegeben werden (vgl. Abschnitt 3-3). Dazu dient das FORTH-Wort **F.R.** Es hat folgende Stack-Relation:

f n --> (5-37)

Der oberste Stack-Eintrag, eine einfach genaue Integer, gibt die Feldbreite an. Er wird gefolgt von der Gleitkommazahl, die das Wort **F.R** ausgeben soll. Das Wort entfernt beide Zahlen vom Stack und gibt die Gleitkommazahl in einem Feld von der gewünschten Breite aus.

5.7.2 Die Grundrechenarten

Die FORTH-Wörter für die Grundrechenarten mit Gleitkommazahlen (Addition, Subtraktion, Multiplikation und Division) lauten **F+**, **F-**, **F*** und **F/**. Sie entsprechen den bereits bekannten arithmetischen Wörtern **+**, **-**, ***** und **/**. Alle haben sie die folgende Stack-Relation:

$f_1\ f_2$ --> f_{erg} (5-38)

Jede der Operationen entfernt also die beiden Gleitkommazahlen vom Stack und ersetzt sie durch eine neue Gleitkommazahl. Beachten Sie, daß bei Gleitkommazahlen eine dem **MOD**-Wort analoge Operation unnötig ist, da die Ergebnisse von Gleitkommaoperationen sowieso mit Nachkommastellen versehen sind. Wenn wir z.B. folgendes eingeben:

```
% 50 % 3 F/ (RETURN)
```

dann erhalten wir als Ergebnis auf dem Stack eine Zahl, die der Gleitkommazahl .166667 E 10 entspricht.

Auch Gleitkommazahlen können in Programmen mit Integers gemischt werden; es liegt jedoch in der Verantwortung des Programmierers, den richtigen Zahlentyp bei der Arbeit an das richtige FORTH-Wort zu übergeben. (Es gibt nämlich keine Möglichkeit, einem Stack-Eintrag anzusehen, welchem Datentyp er zugehört.) So können z.B. Gleitkommaoperationen innerhalb einer Schleife ausgeführt werden. In diesem Fall wären die Schleifenparameter ganzzahlig, während die Operanden der Operationen Nachkommastellen aufweisen.

5.7.3 Stack-Manipulation

Gleitkommazahlen benötigen ebenso wie doppelt genaue Integers zwei Stack-Einträge zur Speicherung. Wenn wir also Gleitkommazahlen auf dem Stack manipulieren wollen, dann können wir dazu dieselben FORTH-Wörter benutzen, die wir für die Manipulation von doppelt genauen Integers bereits kennengelernt haben. Es sind dies **2DUP, 2DROP, 2SWAP** und **2ROLL**. Als Beispiel schreiben wir eine neue Version der bereits bekannten Fakultätsfunktion, diesmal jedoch soll mit Gleitkommazahlen gerechnet werden. Sie sehen das Programm in Abbildung 5-4.

```
0 ( Berechnung der Fakultaet mit Gleitkommazahlen )
1 : FFACT   1  +  %  1      1   4 ROLL   SWAP
2             DO   I   I-F   F*    LOOP  F.        ;
3
4
```

ABBILDUNG 5-4: Fakultätsberechnung mit Gleitkommazahlen

Dieses Programm ist in seinem Aufbau dem von Abbildung 5-1 sehr ähnlich. Der einzige Unterschied besteht darin, daß jetzt mit Gleitkommazahlen gerechnet wird. In der Definition von **FFACT** taucht ein neues Wort auf: **I-F.** Dieses dient zur Umwandlung einer ganzen Zahl (Integer) in eine Gleitkommazahl ("floating-point number). Es entfernt die oberste Integer auf dem Stack und ersetzt sie durch eine gleichwertige Gleitkommazahl. Seine Stack-Relation:

n --> f (5-39)

Es wird also eine Integer, die eine Stack-Position zur Speicherung benötigt, durch eine Gleitkommazahl mit einem Speicherbedarf von zwei Stack-Positionen ersetzt. Vergleichen wir nun das Programm aus Abbildung 5-1 mit dem neuen Programm. In Zeile 2 der Abbildung 5-1 legen wir die 1 als doppelt genaue Integer auf den Stack, indem wir der Reihe nach 1 und 0 pushen. Entsprechend dient in Zeile 2 der Abbildung 5-4 die Eingabe von % 1 zum Ablegen der 1 als Gleitkommazahl auf dem Stack. In Zeile 2 der Abbildung 5-1 legen wir mittels **I** und 0 den Schleifenindex als doppelt genaue Integer auf den Stack. Zeile 3 der Abbildung 5-4 stellt hingegen den Schleifenindex durch die Wörterfolge **I** und **I-F** als Gleitkommazahl zur Verfügung. In Zeile 3 der Abbildung 5-1 drucken wir das doppelt genaue Ergebnis mit dem Wort **D**, während ähnlich in Zeile 3 der Abbildung 5-4 das Ausgabewort **F.** verwendet wird. Die Wörter, die die Stack-Manipulationen besorgen, sind in beiden Programmen gleich. Dies kommt daher, weil beide Programme mit Zahlen arbeiten, die jeweils zwei Stack-Positionen zur Speicherung benötigen.

5.7.4 Gleitkommafunktionen

Neben den Grundrechenarten gibt es noch eine Reihe weiterer mathematischer Funktionen, die auf Gleitkommazahlen angewendet werden können. Wir stellen zuerst einige Funktionen vor, die in ihrer Arbeitsweise bereits bekannten ähneln.

FABS und **FMINUS** - Das FORTH-Kommando **FABS** liefert den Absolutwert einer Gleitkommazahl und ist darin dem bereits bekannten Kommando

ABS ähnlich. **FABS** entfernt eine Gleitkommazahl vom Stack und ersetzt sie durch ihren Absolutwert (als Gleitkommazahl).

Das FORTH-Wort **FMINUS** dient ähnlich wie **NEGATE** zur Umkehrung des Vorzeichens einer Gleitkommazahl. Die beiden Wörter **FABS** und **FMINUS** haben die Stack-Relation

$$f_1 \text{ --> } f_2 \tag{5-40}$$

Sie ersetzen also eine Gleitkommazahl auf dem Stack durch eine andere.

Weiterhin gibt es eine Anzahl von FORTH-Wörtern, die zum Vergleich von Gleitkommazahlen dienen. Das Wort **SGN** erlaubt es, das Vorzeichen einer Gleitkommazahl zu bestimmen. Das Wort entfernt die oberste Gleitkommazahl vom Stack und ersetzt sie durch eine einfach genaue Integer, die als Flag interpretiert werden kann. Der Wert von **SGN** ist entweder -1, 0 oder 1, je nachdem, ob die Ausgangszahl kleiner Null, gleich Null oder größer Null war. **SGN** besitzt folgende Stack-Relation:

$$f \text{ --> } n \tag{5-41}$$

Zum Vergleich zweier Gleitkommazahlen dient das FORTH-Kommando **FCOMP**. Es entfernt zwei Gleitkommazahlen vom Stack und ersetzt sie durch eine einfach genaue Integer. Diese hat entweder den Wert -1,0 oder 1. Wir haben folgende Stack-Relation:

$$f_1 \; f_2 \text{ --> } n \tag{5-42}$$

Dabei ist n gleich -1, falls f_1 kleiner f_2 ist, es hat den Wert 0, falls gilt $f_1 = f_2$ und ist ansonsten - wenn also f_1 größer als f_2 - gleich 1.

Arithmetische Operationen im gemischten Modus sind im Zusammenhang mit Gleitkommazahlen nicht möglich. Man kann also beispielsweise keine Integer mit einer Gleitkommazahl durch eine spezielle gemischte Arithmetikoperation multiplizieren lassen. Es gibt jedoch einen Ausweg aus dieser Notlage. Dieser besteht darin,

Umwandlungsfunktionen anzuwenden, die Zahlen von einem Datentyp in einen anderen umwandeln, also z.B. aus Integers Gleitkommazahlen machen und umgekehrt.

Das Wort **I-F** haben wir bereits kennengelernt. Es wandelt eine Integer in eine Gleitkommazahl um, erwartet also eine einfach genaue Integer an oberster Stack-Position, entfernt sie von dort und ersetzt sie durch eine numerische gleichwertige Gleitkommazahl. Wir wiederholen noch einmal die Stack-Relation:

n --> f (5-43)

In der umgekehrten Richtung arbeitet **CINT.** Die Gleitkommazahl an oberster Stack-Position wird entfernt und durch eine ganze Zahl ersetzt. Dabei handelt es sich um die größte ganze Zahl, die kleiner oder gleich der Ausgangs-Gleitkommazahl ist. Da **CINT** zur Umwandlung von Gleitkommazahlen in ganze Zahlen dient, hat es folgende Stack-Relation:

f --> n (5-44)

Manchmal kann es nötig sein, eine Gleitkommazahl ganzzahlig zu machen, ohne ihren Datentyp zu ändern (d.h., ohne sie in eine Integer umzuwandeln). In diesem Fall stehen zwei FORTH-Wörter zur Verfügung. Eines davon ist **FIX.** Das Wort entfernt die oberste Gleitkommazahl vom Stack und schneidet einfach deren Nachkommastellen ab. Die sich daraus ergebende Gleitkommazahl wird als Ergebnis auf den Stack gelegt.

Auch das FORTH-Wort **INT** arbeitet mit einer Gleitkommazahl, die es durch eine andere Gleitkommazahl ersetzt, welche numerisch gleich der größten ganzen Zahl ist, die kleiner oder gleich der Ausgangszahl ist. Bei positiven Zahlen liefern **FIX** und **INT** gleiche Ergebnisse. Anders sieht die Sache aus, wenn wir mit negativen Werten arbeiten. Beachten Sie, daß -8 die größte ganze Zahl ist, die kleiner oder gleich der Zahl -7,3 ist.

Auch für die mehr fortgeschrittenen mathematischen Funktionen gibt es FORTH-Wörter, die wir jetzt darstellen wollen. Die meisten von ihnen ersetzen eine Gleitkommazahl auf dem Stack durch eine andere.

LOG und **LOG10** - Diese beiden Kommandos dienen zur Berechnung des natürlichen bzw. ds Zehner-Logarithmus einer Zahl. So erhalten wir z.B. durch Eingabe von

% 100 LOG10 (RETURN)

als Ergebnis die Gleitkommazahl 2 auf dem Stack.

EXP und **10^** - Diese beiden FORTH-Kommandos bilden Potenzen der Eulerschen Zahl E, bzw. zur Basis 10. Durch Eingabe von

% 2 10^ (RETURN)

erhalten wir so z.B. die Gleitkommazahl 100 auf dem Stack. (Anmerkung: Auf den meisten Mikrocomputer-Tastaturen wird der Pfeil nach oben auch als Dächlein dargestellt).

1/X und **SQR** - Mit diesen FORTH-Befehlen kann man den Kehrwert bzw. die Quadratwurzel einer Gleitkommazahl berechnen. Beide Wörter entfernen eine Gleitkommazahl vom Stack und ersetzen sie durch ihren Kehrwert (**1/X**) bzw. durch ihre Quadratwurzel (**SQR**).

SIN,COS,TAN und **ATAN** - Diese FORTH-Kommandos berechnen die trigonometrischen Funktionen Sinus, Cosinus, Tangens und Arcustangens. Das Argument dieser Funktionen (die oberste Gleitkommazahl auf dem Stack) kann entweder in Winkelgraden oder im Bogenmaß angegeben werden. Dazu gibt es zwei sehr nützliche FORTH-Wörter, nämlich **RADIANS** und **DEGREES**. Diese beiden Wörter lassen den Stack unverändert. Wenn Sie das Kommando **DEGREES** geben, dann werden die Eingaben zu und die Ausgaben von allen trigonometrischen Funktionen als Winkelgrade interpretiert. Nach Ausführung von **RADIANS** werden die entsprechenden Angaben im Bogenmaß eingelesen bzw. ausgegeben.

RND - Dieses Wort entfernt die oberste Gleitkommazahl vom Stack und legt statt dessen dort eine Pseudozufallszahl ab. Die Zufallszahl ist kleiner oder gleich der Ausgangszahl, vorausgesetzt, die Ausgangszahl war gleich oder größer 1. Die Zufallszahl hat zwar einen ganzzahligen Wert, wird jedoch als Gleitkommazahl

gespeichert. Falls die Ausgangszahl auf dem Stack zwischen 0 und 1 (aber kleiner 1) war, dann ist die Pseudozufallszahl eine reelle Zahl zwischen 0 und 1. Seien Sie vorsichtig, wenn Sie **RND** mit Gleitkommazahlen einsetzen, da sich hier die Ergebnisse von **RND** zusammen mit Integers unterscheiden (vgl. Abschnitt 2-7).

Neben den mathematischen Operationen, die - wie die eben besprochenen - nur ein einziges Argument haben, gibt es natürlich auch mehrstellige Operationen. Diese entfernen 2 Gleitkommazahlen vom Stack und ersetzen sie durch eine einzige.

X^Y - Mit diesem FORTH-Wort können Potenzen von Gleitkommazahlen gebildet werden. **X^Y** ersetzt zwei Gleitkommazahlen auf dem Stack durch eine neue Gleitkommazahl, die sich ergibt, wenn man die zweite Ausgangszahl zu einer Potenz erhebt, die durch die erste Ausgangszahl angegeben ist. Entsprechend lautet die Stack-Relation:

$$f_1 \ f_2 \ \text{-->} \ f_1^{\ f_2} \qquad (5\text{-}45)$$

ATN2 - Diese Funktion dividiert die zweite Gleitkommazahl auf dem Stack durch die erste Gleitkommazahl auf dem Stack und legt dort den Arcustangens des Ergebnisses ab. Die beiden Ausgangszahlen werden natürlich zuvor vom Stack entfernt.

Die Gleitkommaoperationen und -funktionen, die wir hier kennengelernt haben, sind Bestandteil des MMSFORTH-Systems. Denken Sie daran, daß sie nicht zum Standard von FORTH-79 gehören und in anderen Systemen unter Umständen etwas anders lauten können.

5.8 Doppelte Genauigkeit und komplexe Zahlen

Dieser Abschnitt behandelt Zahlen, deren Speicherung vier Stackpositionen benötigt. Das sind einmal doppelt genaue Gleitkommazahlen. Die Gleitkommazahlen des letzten Abschnittes sind in der Regel auf sechs Stellen genau. Dabei ist die Genauigkeit der Zahl nicht von ihrer absoluten Größe abhängig. So haben z.B.

1,23456
123,456
0,123456 E15

allesamt sechs signifikante Ziffern, sie unterscheiden sich aber gewaltig in ihrem Betrag. Beim Rechnen mit Gleitkommazahlen ergeben sich in der Regel kleine Fehler, sog. Rundungsfehler. Einige Programme führen nun eine große Menge von arithmetischen Operationen mit Gleitkommazahlen aus, so daß sich diese Rundungsfehler häufen; der einzelne Rundungsfehler mag ohne Bedeutung sein, der Gesamteffekt kann sich jedoch durchaus auf die Genauigkeit des Ergebnisses bedeutend auswirken. Um mit mehr als sechs Stellen Genauigkeit rechnen zu können, stellen einige Systeme doppelt genaue Gleitkommazahlen zur Verfügung. In der Darstellung unterscheiden sich solche Zahlen dadurch, daß der Exponent von der Mantisse durch den Buchstaben D (und nicht, wie bei einfach genauen Zahlen durch E) abgetrennt wird. Bei der Speicherung benötigen doppelt genaue Gleitkommazahlen vier Stack-Wörter, also genau doppelt soviel wie die einfach genauen Gleitkommazahlen. Deshalb sind Berechnungen mit diesem Zahlentyp auch wesentlich langsamer und verbrauchen mehr Speicherplatz. Sie sollten sich deshalb genau überlegen, ob Sie in Ihrem Programm mit doppelt genauen Zahlen arbeiten wollen. Auch ist dieser Datentyp kein Bestandteil von FORTH-79, findet sich jedoch in MMSFORTH.

Beim Eingeben einer doppelt genauen Gleitkommazahl müssen wir den Interpreter auf den Datentyp der Zahl hinweisen; dies tun wir, indem wir der Zahl die Zeichenfolge **D%** voranstellen. Erinnern Sie sich daran, daß zur Eingabe von einfach genauen Gleitkommazahlen das Prozentzeichen dient (vgl. Abschnitt 5-6). Im allgemeinen werden FORTH-Wörter zur Manipulation von doppelt genauen Gleitkommazahlen gebildet, indem man den Buchstaben D den FORTH-Wörtern voranstellt, die Berechnungen mit einfach genauen Gleitkommazahlen anstellen. So ist z.B. **DF#IN** analog zu **F#IN**. Die Stack-Relation für **DF#IN** lautet:

$$\text{-->}\ d_f \qquad (5\text{-}46)$$

Wie Sie sehen können, stellen wir in Stack-Relationen eine doppelt genaue Gleitkommazahl durch "d_f" dar.

5.8.1 Stack-Manipulationen

Doppelt genaue Gleitkommazahlen benötigen bekanntermaßen vier Stack-Positionen. Deshalb braucht man spezielle FORTH-Wörtern, um mit diesen Zahlen Stack-Manipulationen ausführen zu können. Die Kommandos lauten **4DUP**, **4DROP**, **4SWAP** und **4ROLL**. Sie entsprechen den bekannten Kommandos für einfach genaue Gleitkommazahlen, außer daß die 2 durch eine 4 ersetzt wird. Beispielsweise dupliziert das Kommando **4DUP** die oberste doppelt genaue Gleitkommazahl auf dem Stack und hat deshalb folgende Stack-Relation:

$$d_f\ \text{-->}\ d_f d_f \qquad (5\text{-}47)$$

Es wird also eine Gruppe von vier zusammenhängenden Stack-Wörtern dupliziert.

Das FORTH-Kommando **4DROP** entfernt die oberste doppelt genaue Gleitkommazahl vom Stack. Das bedeutet, daß alle unterhalb liegenden Einträge um 4 Positionen nach oben wandern. Hier die Stack-Relation:

$$d_f\ \text{-->} \qquad (5\text{-}48)$$

Das FORTH-Kommando **4SWAP** vertauscht die beiden obersten doppelt genauen Gleitkommazahlen und hat deshalb folgende Stack-Relation:

$$d_{f1}\ d_{f2}\ \text{-->}\ d_{f2}\ d_{f1} \qquad (5\text{-}49)$$

Das FORTH-Kommando **4OVER** dupliziert die zweite doppelt genaue Gleitkommazahl auf dem Stack an die erste Position. Dies bedeutet, daß die fünfte bis achte Stack-Zeile noch einmal von Anfang an wiederholt werden. Die Stack-Relation ist

$$d_{f1}\ d_{f2} \dashrightarrow d_{f1}\ d_{f2}\ d_{f1} \qquad (5\text{-}50)$$

DF. und **DF.R** - Diese beiden Kommandos entsprechen den Worten **F.** und **F.R.** Das Wort **DF.** dient also zur Ausgabe einer doppelt genauen Gleitkommazahl, wobei diese auch noch vom Stack entfernt wird. Um eine solche Zahl in einem Datenfeld mit vorgegebener Breite auszugeben, benutzt man das Kommando **DF.R,** das eine einfach genaue Integer und eine doppelt genaue Gleitkommazahl vom Stack entfernt. Die Integer dient in gewohnter Weise zur Angabe der Feldbreite. **DF.R** erwartet, daß die Integer an erster und die auszugebende Gleitkommazahl an zweiter Stack-Position zu finden ist.

```
 0 ( Fakultaetsberechnung mit doppelt genauen Gleitkommazahlen )
 1 : DFFACT  1  +     D%  1     1   6 ROLL   SWAP
 2         DO   I   I-F  FDF   DF*    LOOP  DF.        ;
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

ABBILDUNG 5-5: Ein FORTH-Wort zur Berechnung der Fakultät mit doppelt genauen Gleitkommazahlen

5.8.2 Die Grundrechenarten

Die FORTH-Wörter für die Addition, Subtraktion, Multiplikation und Division mit doppelt genauen Gleitkommazahlen lauten **DF+**, **DF-**, **DF** und **DF/**. Sie entsprechen den bereits bekannten Kommandos **F+**,**F-**, **F*** und **F/**, außer daß die Operanden jeweils vier Stack-Positionen anstelle von zwei beanspruchen. Alle haben sie die folgende Stack-Relation gemeinsam:

d_{f1} d_{f2} --> d_{ferg} (5-51)

5.8.3 Typumwandlung

Auch für die Umwandlung zwischen einfach genauen und doppelt genauen Gleitkommazahlen gibt es spezielle Kommandos in FORTH. Zur Umwandlung einer einfach genauen Gleitkommazahl auf dem Stack in eine doppelt genaue dient das Kommando **FDF**. Seine Stack-Relation sieht folgendermaßen aus:

f --> d_f (5-52)

Den umgekehrten Weg kann man mit **DFF** gehen. Dieses Wort entfernt eine doppelt genaue Gleitkommazahl vom Stack und legt an ihrer Stelle eine einfach genaue Gleitkommazahl dort ab. Das Ergebnis wird abgerundet, da es weniger signifikante Stellen aufweisen kann. Die Stack-Relation ist

d_f --> f (5-53)

Das, was wir über doppelt genaue Gleitkommazahlen kennengelernt haben, wollen wir noch einmal verdeutlichen, indem wir das Beispielprogramm 5-4 neu schreiben (vgl. Abbildung 5-5). Wir gehen hier nur auf die Unterschiede zwischen diesen beiden Programmen ein. In Zeile 2 benutzen wir anstelle von **%** das FORTH-Kommando

D%. Auf diese Art wird die 1 als doppelt genaue Gleitkommazahl auf den Stack gelegt. Ebenso findet sich in Zeile 2 anstelle von 4 **ROLL** die Befehlsfolge 6 **ROLL.** Dies kommt daher, weil eine doppelt genaue Gleitkommazahl zwei Stack-Positionen mehr an Speicherplatz benötigt, als eine einfach genaue Gleitkommazahl. In Zeile 2 wollen wir wieder den Schleifenindex in das passende Datenformat umwandeln. Dazu bedienen wir uns zuerst des Kommandos **I-F.** Dadurch erhalten wir eine einfach genaue Gleitkommazahl, die wir jetzt mittels **FDF** in eine doppelt genaue umwandeln. Das restliche Programm entspricht weitestgehend dem in der Abbildung 5-4, außer daß zur Multiplikation und Ausgabe des Ergebnisses die Wörter **DF*** und **DF.** benutzt werden.

Eine Handvoll Funktionen für doppelt genaue Gleitkommazahlen entspricht den bereits bekannten Funktionen für einfach genaue Zahlen dieses Typs. Sie lauten **DFABS, DFMINUS, DFSIGN** und **DFCOMP.** Sie entsprechen jeweils den Funktionen **FABS, FMINUS, FSIGN** und **FCOMP.** Im übrigen steht eine wesentlich größere Anzahl mathematischer Funktionen für einfach genaue als für doppelt genaue Gleitkommazahlen zur Verfügung.

5.8.4 Komplexe Zahlen

Komplexe Zahlen setzen sich aus einem Realteil und einem Imaginärteil zusammen. MMSFORTH sieht Kommandos vor, mit denen auch dieser Zahlentyp manipuliert werden kann. Jede komplexe Zahl besteht aus einem Paar einfach genauer Gleitkommazahlen. Deshalb belegen auch komplexe Zahlen 4 Stack-Positionen, weswegen man zur Stack-Manipulation bei diesem Datentyp die Kommandos **4DUP, 4DROP, 4SWAP** und **4OVER** heranziehen kann.

Man signalisiert dem FORTH-Interpreter durch die Zeichenfolge **CP%**, daß eine komplexe Zahl eingegeben werden soll. Umgekehrt sorgt das FORTH-Wort **CP.** dafür, daß eine komplexe Zahl vom Stack entfernt und ausgegeben wird. Dabei wird davon ausgegangen, daß der Imaginärteil der Zahl zuoberst auf dem Stack liegt, gefolgt vom Realteil.

Zum Beispiel erreicht man durch Eingeben von

```
CP% 5 7 CP. (RETURN)
```

daß folgendes ausgegeben wird:

(5,7)

Für die Arithmetik mit komplexen Zahlen stehen die Wörter **CP+**, **CP-**, **CP*** und **CP/** zur Verfügung. Alle vier entfernen zwei komplexe Zahlen vom Stack und ersetzen diese durch das Ergebnis (die Summe, Differenz, das Produkt oder den Quotienten). Ihre Stack-Relation lautet daher:

$$c_{p1}\ c_{p2} \text{ --> } c_{perg} \qquad (5\text{-}54)$$

Wie Sie sehen können, stellen wir komplexe Zahlen in den Stack-Relationen mittels "c_p" dar. Denken Sie daran, daß auch dieser Datentyp vier Stack-Positionen benötigt.

Die Wörter **MAG** und **PHASE** entfernen eine komplexe Zahl vom Stack und ersetzen sie durch eine einfach genaue Gleitkommazahl, die entweder gleich dem Betrag oder der Phase (dem Winkel) der komplexen Zahl ist. Bei der Darstellung der Phase kann man zwischen Winkelgraden und Bogengraden wählen. Dies erreichen Sie, indem Sie auf die bereits bekannten Wörter **DEGREES** oder **RADIANS** zurückgreifen (vgl. Abschnitt 5-6).

Die Wörter **R-P** und **P-R** dienen dazu, komplexe Zahlen von einem Darstellungsformat ins andere umzuwandeln. Bei Ausführung von **R-P** sollte die komplexe Zahl, die an oberster Stack-Position steht, in cartesischen Koordinaten gegeben sein, wobei sich der Imaginärteil an oberster Stack-Position befindet. **R-P** ersetzt diese Zahl durch ihre Darstellung in Polarkoordinaten (d.h., Betrag und Winkel). Dabei wird der Winkel an oberste Stack-Position gelegt. Der Realteil und Imaginärteil bei der cartesischen Darstellung bzw. der Betrag und der Winkel bei der Polardarstellung sind jeweils einfach genaue Gleitkommazahlen. Sie sollten hier mit Vorsicht vorgehen. Es gibt ja keine Möglichkeit, nur durch Inspizieren des Stacks festzustellen, welche Art von Zahlen auf ihm gespeichert sind. So kann man z.B. eine einzelne komplexe Zahl nicht von vier einfach genauen Integers unterscheiden. Die mathematischen Operationen **CP+**, **CP-**, **CP*** und **CP/** nehmen allesamt an, daß ihre Argumente, die komplexen Zahlen, in cartesischer Form gegeben sind (also in der Form Realteil, Imaginärteil). Deshalb sollte man die Umwandlungsfunktionen **P-R** und **R-P** nur bei der Ein-

Ausgabe von Daten anwenden, um hier zwischen diesen beiden Darstellungsmöglichkeiten wählen zu können.

Noch zwei weitere FORTH-Wörter sind im Zusammenhang mit komplexen Zahlen verfügbar: **CPMINUS** und **CONJ.** Diese ersetzen jeweils eine komplexe Zahl an oberster Stack-Position durch ihren negativen Wert (**CPMINUS**) bzw. durch ihre komplex konjugierte Zahl (**CONJ**).

5.9 Übungsaufgaben

In vielen der folgenden Übungsaufgaben werden Sie gebeten, FORTH-Wörter oder Programme zu schreiben. Überprüfen Sie Ihre Ergebnisse auf Ihrem eigenen Computer. Achten Sie darauf, die Definitionen möglichst kurz zu halten, indem Sie eine komplizierte Aufgabe in mehrere Unterwörter "aufbrechen" (Modularisierung).

5-1 Erörtern Sie den Unterschied zwischen einfach genauen und doppelt genauen Integers.

5-2 Wiederholen Sie Aufgabe 4-8 mit doppelt genauen Integers.

5-3 Wiederholen Sie Aufgabe 4-9 mit doppelt genauen Integers.

5-4 Wiederholen Sie Aufgabe 5-3, lassen Sie das Programm jetzt aber abbrechen, wenn die Summe größer 200 000 wird.

5-5 Wiederholen Sie Aufgabe 4-13 mit doppelt genauen Integers; lassen Sie das Programm mit einer Fehlermeldung abbrechen, wenn das Produkt eine Million übersteigt.

5-6 Wiederholen Sie Aufgabe 4-11 mit doppelt genauen Integers, und geben Sie das Ergebnis in einem 20 Zeichen breiten Datenfeld aus.

5-7 Schreiben Sie ein FORTH-Wort, das die Funktion

$$ab/(c+d)$$

möglichst genau berechnet. Die Variablenwerte a, b, c und d sollen auf dem Stack zur Verfügung gestellt werden.

5-8 Wiederholen Sie Aufgabe 5-7 mit doppelt genauen Integers.

5-9 Schreiben Sie ein FORTH-Wort, das unter Verwendung von doppelt genauen Integers den Punktedurchschnitt eines Studenten in fünf Prüfungen berechnet. Der Durchschnittswert soll mit dreistelliger Genauigkeit hinter dem Dezimalpunkt ausgegeben werden. Vergessen Sie nicht, das Dezimalkomma zu drucken!

5-10 Schreiben Sie ein FORTH-Wort zur Berechnung der Verkäufer-Provision nach folgender Vorschrift: Bei Umsätzen bis 10 000 DM beträgt die Provision 10%, bei Umsätzen zwischen 10 000 und 50 000 DM beträgt die Provision 15%. Übersteigen die Umsätze 50 000 DM, so erhält der Verkäufer 18% Provision. Das FORTH-Wort soll die Summe der einzelnen Verkäufe berechnen und darauf basierend den Provisionsbetrag. Das Ergebnis soll als Geldbetrag ausgegeben werden, d.h., die Zeichenfolge DM und ein Dezimalkomma müssen an passender Stelle erscheinen. Außerdem sollten Sie Tausenderstellen im Betrag durch ein Leerzeichen von den restlichen Ziffern abteilen. Sie können davon ausgehen, daß der Provisionsbetrag des Verkäufers 100 000 DM nicht übersteigt. Arbeiten Sie mit doppelt genauen Integers. Versuchen Sie, Ihr Programm zu modularisieren, d.h., die Lösung in mehrere FORTH-Wörter aufzuteilen.

5-11 Schreiben Sie ein Programm zur Berechnung der pythagoreischen Zahlentripel mit doppelt genauen Integers.

5-12 Schreiben Sie ein FORTH-Wort zur Berechnung des folgenden Ausdrucks:

$$ab/(c+d) + e$$

wobei a, b, c, d und e Integers sind. Die Zwischenergebnisse Ihrer Berechnungen sollten in doppelter Genauigkeit abgelegt werden.

5-13 Erörtern Sie die Vorteile von Rechenoperationen im gemischten Modus.

5-14 Was bedeutet der Ausdruck "vorzeichenlose Integer"?

5-15 Kann auch eine vorzeichenlose Integer von doppelter Genauigkeit sein?

5-16 Schreiben Sie ein FORTH-Wort zur Berechnung der pythagoreischen Zahlentripel mit vorzeichenlosen einfach genauen Integers.

5-17 Wiederholen Sie Aufgabe 5-16, arbeiten Sie diesmal jedoch mit doppelt genauen Integers. Überlegen Sie sich genau, wie Sie die Daten ausgeben.

5-18 Erörtern Sie die Funktionsweise der FORTH-Kommandos **AND**,**OR** und **XOR**. Schreiben Sie eigene Programme mit diesen Worten.

5-19 Legen Sie den Unterschied zwischen Gleitkommazahlen und ganzen Zahlen dar.

5-20 Wiederholen Sie Aufgabe 5-10 mit Gleitkommazahlen anstelle von Integers.

5-21 Wiederholen Sie Aufgabe 5-11 mit Gleitkommazahlen.

5-22 Wiederholen Sie Aufgabe 5-20 mit doppelt genauen Gleitkommazahlen.

5-23 Wiederholen Sie Aufgabe 5-21 mit doppelt genauen Gleitkommazahlen.

5-24 Schreiben Sie ein FORTH-Wort zur Berechnung des folgenden Ausdrucks

$$(a+b)c/d$$

wobei a, b, c und d komplexe Zahlen sind.

6

Konstanten, Variable und Arrays

6 Konstanten, Variable und Arrays

In allen Beispielprogrammen dieses Buches wurden Daten auf den Stack übergeben. Dies ist guter FORTH-Programmierstil, da es Speicherplatz spart und dafür sorgt, daß Programme schneller laufen. Manche Aufgabenstellungen erfordern jedoch eine andere Speicherungsmethode, die für den Programmierer bequemer ist. Stellen Sie sich nur einmal vor, Sie wollen eine Konstante in einem Programm mehrfach verwenden. Vor allem dann, wenn diese Konstante viele Stellen aufweist, ist es äußerst mühselig, sie jedesmal einzugeben, wenn Sie sie brauchen. FORTH gibt Ihnen die Möglichkeit, solchen Konstanten symbolische Namen zuzuweisen und sie im Arbeitsspeicher des Rechners abzulegen. Nachdem ein Name für die Konstante vereinbart ist, kann man einfach den Namen anstelle des Konstantenwerts in den Programmen eingeben. Andere Anwendungen erfordern den Einsatz von sog. Variablen, also Größen, deren Werte im Programm erst berechnet werden. Natürlich kann man solche Variable auch auf dem Stack speichern. Verwendet ein Programm aber sehr viele Variable, dann ist es für den Programmierer sehr umständlich, über jede dieser Variablen auf dem Stack Buch zu führen. FORTH stellt auch hier Lösungsverfahren bereit, mit denen Variable benannt und im Arbeitsspeicher des Computers gespeichert werden können. Mit einem einfachen Verfahren kann der Programmierer die Variablenwerte in seinem Programm erreichen, indem er auf sie über den Variablennamen zugreift. Auch ist es sehr einfach, den Wert einer Variablen zu ändern. Wir können also einen Variablenwert berechnen, im Speicher ablegen, diese Variable in Berechnungen einsetzen, anschließend den Wert der Variablen verändern und den ganzen Prozeß wiederholen. Diesem Verfahren gilt unsere Aufmerksamkeit im vorliegenden Kapitel.

Oft arbeiten Programme mit ganzen Gruppen solcher Variablenwerte. Dies kann z.B. nötig sein, wenn wir für alle Teilnehmer eines Kurses die Punktzahl in Prüfungen ausrechnen. Zwar ist es möglich, für jeden Teilnehmer das Programm getrennt laufen zu lassen; gerade dann, wenn in der Klasse viele Schüler sind, ist das Verfahren aber viel zu umständlich. Wir werden Ihnen in diesem Kapitel zeigen, wie solche Variablengruppen zu sog. Arrays zusammengefaßt werden können und wie man auf eine einfache und elegante Weise mit diesen in Programmen arbeitet. Doch zunächst stellen wir Ihnen die Konstanten vor.

6.1 Konstanten

Wir werden jetzt ein Verfahren diskutieren, mit dem Konstanten im Arbeitsspeicher des Computers unter einem beliebigen, frei wählbaren Namen gespeichert werden können. Sie können dann diese Konstante auf den Stack bekommen, indem Sie einfach ihren Namen eingeben. Konstanten sind Werte, die sich während der Ausführung des Programms nicht ändern. Es gibt zwar Möglichkeiten zur Änderung eines Konstantenwerts während des Programmlaufs; ihre Handhabung ist jedoch umständlich, und wir raten von ihrem Gebrauch ab. Wenn Sie in Ihren Berechnungen sich verändernde Werte unter bestimmten Namen abspeichern wollen, dann sollten Sie dies über Variable tun. Will man den Wert einer Konstanten in einem Programm ändern, so kann man dies durch Edieren des Programms erreichen. Darüber später mehr in diesem Kapitel.

CONSTANT - Zum Definieren einer Konstante dient das FORTH-Wort **CONSTANT**. Wir benutzen es in folgender Form:

```
3600 CONSTANT H/S (RETURN)                                  (6-1)
```

Damit haben wir der Konstante mit dem Namen **H/S** den Wert 3600 zugewiesen. Wir können jetzt überall dort, wo wir den Wert 3600 benötigen, auch den Namen **H/S** einsetzen. Bei Ausführung des FORTH-Befehls **CONSTANT** wird die einfach genaue Integer an oberster Stack-Position - in unserem Beispiel 3600 - vom Stack entfernt und im Wörterbuch des Systems gespeichert, welches sich im Arbeitsspeicher des Computers befindet. Als Name für die Konstante wird der Ausdruck verwendet, der unmittelbar auf den Befehl **CONSTANT** folgt. Die Stack-Relation lautet:

```
n -->                                                       (6-2)
```

Als Beispiel sehen Sie in Abbildung 6-1 ein FORTH-Wort zur Umwandlung von Stunden in Sekunden. Zeile 1 definiert die Konstante **H/S** so, wie Sie es aus Beispiel 6-1 kennen. In Zeile 2 wird ein neues FORTH-Wort vereinbart, das den Namen **STD/SEC** trägt. Dieses Wort soll nichts anderes machen, als die Integer an oberster Stack-Position mit 3600 zu multiplizieren. Wir wollen also den

wert 3600 auf den Stack legen; dies könnten wir natürlich erreichen, indem wir ihn hinschreiben. Statt dessen benutzen wir jedoch die Konstante **H/S.** Dieses Beispiel ist zwar sehr einfach, läßt jedoch die grundlegenden Prinzipien beim Einsatz von Konstanten erkennen. Eine Konstante wird nicht als Teil einer Wortdefinition vereinbart. Sie müssen sie definieren, ehe Sie sie in einem Programm einsetzen können.

```
0 ( Beispiel fuer den Einsatz von Konstanten )
1 3600 CONSTANT H/S
2 : STD/SEC  H/S   *     .  ;
3
4
```

ABBILDUNG 6-1: Ein Beispiel für den Einsatz von Konstanten

Für den Einsatz von Konstanten gibt es mehrere Gründe. Einer ist, daß Konstanten leichter (kürzer) zu schreiben sind als Zahlen. Ein weiterer Grund besteht darin, daß man sich den Namen einer Konstante leichter merken kann als die Zahl selbst. (Denken Sie z.B. an die Kreiszahl "pi" mit ihrem Wert von 3,1415...) Ein weiterer Grund besteht darin, daß man eine Konstante leicht ändern kann, ehe man das Programm laufen läßt. Nehmen Sie z.B. einmal an, Sie haben ein Programm geschrieben, das Zinsberechnungen ausführt. In diesem Programm kommt der Zinssatz, mit dem gearbeitet werden soll, mehrfach vor. Wenn Sie jetzt den Zinsbetrag als Zahl in Ihr Programm mitaufgenommen haben und er ändert sich, dann müssen Sie mühselig von Hand jedes Vorkommnis des Zinsbetrags abändern. Anders, wenn Sie eine Konstante benutzen. Dort, wo in der ersten Version der tatsächliche numerische Betrag des Zinssatzes stand, steht jetzt nur eine Konstante. Ändert sich der Zinssatz, dann brauchen Sie vor Ausführung des Programms lediglich die Konstante zu ändern und haben sonst keine Arbeit mehr! Nicht zuletzt kann - gerade in Fällen wie dem eben geschilderten - die Verwendung von Konstanten mit dazu beitragen, daß Speicherplatz in den Programmen eingespart wird und diese schneller laufen.

2CONSTANT - Natürlich kann man auch Konstanten mit doppelt genauen Integers definieren. Dazu dient das FORTH-Wort **2CONSTANT.** Dieses verwendet man genauso wie das bereits bekannte **CONSTANT**, außer daß jetzt eine doppelt genaue Zahl definiert wird.

```
65535. 2CONSTANT MEMSIZE (RETURN)                    (6-3)
```

Jetzt können Sie in Ihren Programmen die Konstante **MEMSIZE** anstelle der doppelt genauen Integer 65535. schreiben. Die Stack-Relation für **2CONSTANT** lautet:

```
d -->                                                (6-4)
```

Denken Sie daran, daß in vielen FORTH-Systemen erst spezielle Programmblöcke geladen werden müssen, ehe Sie mit doppelt genauen Integers arbeiten können.

6.1.1 Gleitkommakonstanten

Wenn Ihr FORTH-System mit Gleitkommazahlen rechnen kann, dann wird es höchstwahrscheinlich auch über die Möglichkeit verfügen, Gleitkommakonstanten zu definieren. Wir stellen hier die Wörter dar, die im MMSFORTH dafür zur Verfügung stehen. Um sie benutzen zu können, müssen Sie jedoch den Programmteil für Gleitkomma-arithmetik laden. Denken Sie noch einmal daran, daß es keinen FORTH-Standard für Gleitkommazahlen gibt, so daß die nötigen Befehle von System zu System unterschiedlich lauten können. Für das MMSFORTH-System sind hier die Wörter **2CONSTANT** und **4CONSTANT** maßgeblich. Sie arbeiten mit einfach bzw. doppelt genauen Gleitkommazahlen, welche Sie vom Stack entfernen und im Lexikon unter dem angegebenen Namen speichern. Diese beiden Wörter werden analog wie in Beispiel 6-1 bzw. 6-3 eingesetzt. Das Wort zur Definition einer einfach genauen Gleitkommakonstanten ist das gleiche wie das für doppelt genaue Integers. Warum dies so ist, geht aus den Ausführungen von Abschnitt 2-4 hervor.

6.2 Variable

Eine Variable ist ein Wert, der erst zum Zeitpunkt der Programmausführung berechnet wird. Oftmals kommt es vor, daß dieser Wert an verschiedenen Stellen in einem FORTH-Wort benötigt wird, oder daß auch andere FORTH-Wörter diesen Wert benutzen wollen. Es ist mühselig, den Wert auf den Stack zu speichern und sich stets darüber im klaren zu sein, wo er sich gerade befindet. Deshalb erlaubt es FORTH, solchen Variablen Namen zu geben und sie an anderer Stelle im Arbeitsspeicher abzulegen. In diesem Punkt ähneln die Variablen den im letzten Abschnitt besprochenen Konstanten. Es gibt jedoch einen wesentlichen Unterschied: Variable in FORTH sind so ausgelegt, daß man ihren Wert mit möglichst wenig Aufwand ändern kann. Im Unterschied dazu haben die Entwickler von FORTH bei den Konstanten darauf geachtet, daß man möglichst schnell auf ihren Wert zugreifen kann.

VARIABLE - Mit dem FORTH-Wort **VARIABLE** wird Speicherplatz für eine einfach genaue Integer im Arbeitsspeicher des Computers reserviert und für diesen ein Name vereinbart. Es wird diesem Speicherplatz jedoch noch kein Wert zugewiesen. Darin unterscheidet sich **VARIABLE** von dem FORTH-Befehl **CONSTANT**, der ja nicht nur Speicherplatz reserviert und mit einem Namen versieht, sondern auch einen festen Wert dort ablegt, welcher sich dazu auf dem Stack befinden muß. Ein Beispiel für den Befehl **VARIABLE**:

```
VARIABLE KEE (RETURN)                    (6-5)
```

Dadurch wird im Wörterbuch des Computers Speicherplatz für eine einfach genaue Integer reserviert. Dieser Speicherplatz erhält den Namen **KEE**. Trifft das FORTH-System von jetzt an auf den Namen **KEE**, dann legt es die Adresse dieser Variablen auf den Stack.

! und@ - Mit dem FORTH-Wort ! speichert man einen Wert an einer bestimmten Adresse ab. Meistens hat man für diese Adresse mit dem FORTH-Wort **VARIABLE** zuvor einen Namen vereinbart. Wenn wir z.B. wie in (6-5) eine Variable **KEE** vereinbart haben, dann können wir dieser jetzt den Wert 234 zuweisen, indem wir folgendes eingeben:

```
234 KEE !                                (6-6)
```

Wenn das FORTH-System auf den Variablennamen **KEE** stößt, dann pusht es die Adresse dieser Variablen auf den Stack. Der Befehl **!** nimmt sich diese Adresse von **KEE** und den Wert 234 und entfernt sie vom Stack. Anschließend speichert er die 234 an der angegebenen Adresse. Das Wort **!** hat folgende Stack-Relation:

n a --> (6-7)

Wir verwenden also in Stack-Diagrammen zur Darstellung von Speicheradressen den Buchstaben "a". Wenn der Variablen **KEE** vor Ausführung von (6-6) bereits ein Wert zugewiesen worden war, dann ist dieser jetzt verloren. Nach Ausführung von (6-6) hat die Speicheradresse mit dem Namen **KEE** unwiderruflich den Wert 234, egal, was sich dort zuvor befunden haben mag.

Mit dem FORTH-Kommando @ besorgen wir uns den Wert einer Variablen und legen ihn auf den Stack. Wir gehen wieder davon aus, daß soeben (6-6) ausgeführt und damit der Variablen **KEE** der Wert 234 zugewiesen worden ist. Wir wollen uns jetzt ansehen, welchen Wert **KEE** hat und dazu den Wert auf den Stack bekommen. Dies können wir folgendermaßen bewerkstelligen:

KEE@ (6-8)

Der Befehl hat folgende Stack-Relation:

a --> n (6-9)

Er funktioniert folgendermaßen:

@ entfernt die oberste Zahl vom Stack und behandelt sie als eine Speicheradresse. Der Wert, der unter dieser Adresse abgespeichert ist, wird als nächstes auf den Stack gelegt und ersetzt dort die ursprüngliche Adresse. Diese Operation ändert nichts am Wert der Variablen, d.h., **KEE** hat vor und nach Ausführung von (6-8) unverändert den Wert 234.

Wir wollen den Einsatz von Variablen jetzt an unserem bereits bekannten Beispiel mit den pythagoreischen Zahlentripeln demon-

strieren (vgl. Ab. 6-2). Wir wollen dieses Programm mit dem in Abbildung 4-7 vergleichen, bei dem alle Werte auf dem Stack gespeichert waren.

```
 0 (Pythagoreische Zahlentripel mit Variablen)
   1 VARIABLE A VARIABLE B VARIABLE C
   2 VARIABLE A1  VARIABLE B1  VARIABLE C1
   3 : PT 15 .R  ;
   4 : SQUARE   DUP   *  ;
   5 : VPYTHTRIP   CR  100   1 DO I DUP SQUARE  A !   A1 !
   6       100   I   DO  I  DUP  SQUARE  B !  B1 !
   7          142   I   DO I DUP  SQUARE  C !  C1 !
   8            A @ B @ + C @ -  DUP      0= IF
   9             A1 @  PT   B1 @ PT   C1 @  PT   CR
  10               THEN    0< IF LEAVE THEN
  11            LOOP
  12        LOOP
  13  LOOP  ;
  14
  15
```

ABBILDUNG 6-2: Berechnung pythagoreischer Zahlentripel mit Variablen

In den ersten beiden Zeilen der Abbildung 6-2 vereinbaren wir sechs Variable mit den Namen **A,B,C,A1,B1** und **C1**. Nachdem wir uns in Zeile 3 und 4 die Wörter **PT** und **SQUARE** definiert haben, sind alle nötigen Vorbereitungen abgeschlossen. Wenden wir uns jetzt dem eigentlich interessanten Wort zu, das auf Zeile 5 beginnt und den Namen **VPYTHTRIP** trägt. Das einleitende **CR** steht hier aus ästhetischen Gründen, um die Lesbarkeit unserer Ergebnisse zu steigern. Wieder haben wir es mit drei ineinander geschachtelten Schleifen zu tun. Die äußere wird 100mal durchlaufen, wobei zuerst der Schleifenindex auf den Stack gelegt und dupliziert wird. Als nächstes rufen wir das Wort **SQUARE**. Jetzt enthält der oberste Stack-Eintrag das Quadrat des äußersten Schleifenindex, während der zweite Stack-Eintrag den Index selbst enthält. Nach Ausführung von **A** und **!** wird der oberste Stack-Eintrag in der Variablen **A** gespeichert. Diese enthält nun also das Quadrat des Laufindex der äußersten Schleife. Ähnlich speichern wir uns durch die Befehlsfolge **A1** und **!** in der Variablen **A1** den Index der äußeren Schleife. Nach Ausführung von Zeile 5 ist der Stack leer. Die

Zeilen 6 und 7 wiederholen im wesentlichen dieselben Schritte wie die Zeile 5. Das bedeutet, daß nach Ausführung dieser beiden Zeilen die Variablen **B** und **B1** das Quadrat sowie den Index der mittleren Schleife enthalten. Schließlich speichern wir in **C** und **C1** auch noch die entsprechenden Größen für die innerste Schleife. In Zeile 8 wird der Wert der Variablen **A** und **B** auf den Stack gelegt und anschließend mittels + addiert. Der Stack enthält nun an oberster Stelle den Wert von $a^2 + b^2$. Diesen vergleichen wir mit dem Wert von C, indem wir **C** ausführen. Um herauszufinden, ob $a^2 + b^2 = c^2$ gilt, subtrahieren wir den Wert von C mittels -. Diese Differenz duplizieren wir und unterwerfen sie dann einem Vergleich mittels **0=**. Fällt der Vergleich positiv aus, d.h., das Flag vom Stack ist "wahr", dann besorgen wir uns den Wert der Variablen **A1**,**B1** und **C1** aus dem Arbeitsspeicher und geben ihn aus. Ansonsten überprüfen wir, ob der Wert von $a^2 + b^2 - c^2$ kleiner 0 ist. Ist dies der Fall, dann verlassen wir die innerste Schleife. Wie Sie sehen, benutzen wir hier den gleichen Algorithmus wie in Abbildung 4-7. Beachten Sie, daß die Variablenvereinbarung kein Teil der Wortdefinition ist. Variable müssen stets definiert werden, <u>ehe</u> man sie in einem Wort einsetzen kann.

Variable machen dem Programmierer das Leben wesentlich angenehmer. Wie Sie an dem Beispiel in Abbildung 6-2 sehen können, muß man sich beim Programmieren mit Variablen um wesentlich weniger Details kümmern, als es bei dem vergleichbaren Programm in Abbildung 4-7 nötig war. Dieser Vorteil muß jedoch mit einem Nachteil erkauft werden: Programme mit Variablen benötigen in der Regel eine längere Ausführungszeit als solche, die nur mit dem Stack arbeiten. Dies kommt daher, weil zum Speichern und Wiederauffinden der Variablenwerte zusätzliche Operationen nötig sind. Diese wollen wir jetzt darlegen. Wenn Sie eine Variable definieren, dann wird diese in das Lexikon von FORTH mit aufgenommen. Allerdings unterscheidet sich der Wörterbucheintrag einer Variablen von dem für ein Wort (vgl. Abschnitt 2-4). Der Wörterbucheintrag für ein Wort enthält ja neben dem Namen auch noch die Instruktionen, die seine Definition ausmachen; der Wörterbucheintrag für eine Variable reserviert lediglich Speicherplatz für diese Variable. Wenn FORTH auf einen Variablennamen stößt, dann legt es die Adresse des Wörterbucheintrags dieser Variablen auf den Stack. Der Befehl **!** nimmt diese Adresse und die nächste Zahl auf dem Stack und speichert die Zahl an der angegebenen Adresse. Ähnlich entfernt @ eine Zahl vom Stack und interpretiert diese als die Adresse der Variablen, deren Wert auszugeben ist.

Variablenadressen werden intern als vorzeichenlose einfach genaue Integers gespeichert. Den Befehlen **!** oder @ muß deshalb nicht unbedingt ein Variablenname vorausgehen. Wenn der Programmierer die fragliche Adresse genau kennt, dann kann er sie direkt als vorzeichenlose Integer eingeben.

All diese einzelnen Arbeitsschritte verlangsamen das Programm. Die Wörterbuchsuche braucht ihre Zeit, ebenso das Aufsuchen von Adressen und Speichern von Zahlen an diesen Adressen. Wenn also alle anderen Umstände gleich sind, dann läuft ein FORTH-Programm ohne Variable schneller als eines, das Variable verwendet. Das Programm aus Abbildung 4-7 läuft so z.B. 1,35mal schneller als das Programm in Abbildung 6-2.

FORGET - Dieses bereits aus Kapitel 2-4 bekannte Wort kann für Variable genauso verwendet werden wie für normale FORTH-Wörter.

+! - Oftmals kommt es vor, daß wir auf eine Variable einen bestimmten Integerwert addieren wollen. Wir könnten dazu die Variable holen, die beiden Zahlen addieren und anschließend die Summe wieder speichern, es gibt jedoch ein einziges FORTH-Wort, das all diese Schritte ausführt. Dieses lautet **+!** und macht folgendes: Der zweite Stack-Eintrag, eine einfach genaue Integer, wird zum Wert der Variablen hinzuaddiert, deren Adresse sich an oberster Stack-Position befindet. Beide - Variablenadresse und Integer - werden vom Stack entfernt.

6.2.1 Variable und doppelt genaue Integers

Es ist auch möglich, doppelt genaue Integerwerte in Variablen abzulegen. In den Einzelheiten funktionieren die dazu dienenden Wörter ähnlich wie die für einfach genaue Integers.

2VARIABLE - Das FORTH-Wort **2VARIABLE** ist ähnlich wie **VARIABLE**, reserviert aber im Wörterbuch Speicherplatz für eine doppelt genaue Integer. Mit

```
2VARIABLE KEY                                    (6-10)
```

richten Sie also einen Wörterbucheintrag ein, der den Namen **KEY** hat und eine doppelt genaue Integer aufnehmen kann.

2! und **2@** - Dieses sind die FORTH-Kommandos für das Speichern (**2!**) und Wiederfinden (**2@**) von Variablen bzw. ihren Werten. Das Wort **2!** hat die Stack-Relation

d a --> (6-11)

Die Stack-Relation für **2@** lautet:

a --> d (6-12)

Denken Sie daran, daß in Stack-Relationen "a" zur Darstellung vorzeichenloser einfach genauer Integers dient, während mit "d" eine doppelt genaue Integer signalisiert wird.

6.2.2 Variable und Gleitkommazahlen

Das MMSFORTH-System und andere FORTH-Systeme erlauben es auch, Gleitkommazahlen in Variablen zu speichern. Wir stellen diese Möglichkeiten dar, möchten Sie aber noch einmal daran erinnern, daß die zugehörigen Wörter nicht standardisiert sind. Für die Arbeit mit einfach genauen Gleitkommazahlen verwendet man dieselben FORTH-Kommandos wie bei doppelt genauen Integers, also **2VARIABLE**, **2!** und **2@**.

Beim Einsatz mit Gleitkommazahlen schreiben wir die Stack-Relationen für **2!** und **2@** in folgender Form:

f a --> (6-13a)

und

a --> f (6-13b)

Wenn Sie doppelt genaue Gleitkommavariable benötigen, dann brauchen Sie nur Kommandos zu geben, die denen für einfach genaue Variable entsprechen, außer daß anstelle einer 2 eine 4 steht. Wir haben also die Wörter **4VARIABLE, 4!** und **4@**. Die Stack-Relationen für **4!** und **4@** lauten:

$$d_f \; a \; \text{-->} \qquad (6\text{-}14a)$$

und

$$a \; \text{-->} \; d_f \qquad (6\text{-}14b)$$

6.3 Arrays

Viele Programme führen wiederholte Berechnungen mit einer ganzen Liste von Variablen aus. Wir haben gesehen, daß über Programmschleifen das Wiederholen von Berechnungen drastisch vereinfacht wird. Jetzt werden wir ein weiteres Verfahren kennenlernen, das solche Prozeduren vereinfacht, falls Variablenlisten mit ins Spiel kommen sollen. In der Sprache der Programmierer nennt man diese Variablenlisten auch Arrays.

Aus den letzten Abschnitten wissen Sie, daß durch die Deklaration einer Variablen (durch das Wort **VARIABLE** oder andere) Speicherplatz im Arbeitsspeicher des Computers reserviert wird. Dies wollen wir jetzt etwas genauer untersuchen. In einem typischen Heim- oder Personal Computer werden einfach genaue Integers in 16 Bit gespeichert. Da man in der EDV üblicherweise eine Gruppe von 8 Bit zu einem Byte zusammenfaßt, kann man sagen, daß zur Speicherung von einfach genauen Integers zwei Byte benötigt werden. Für doppelt genaue Integers braucht man entsprechend vier Byte. Üblicherweise haben die Speicherwörter eines Computers (die kleinste vom Computer adressierbare Einheit im Arbeitsspeicher) ebenfalls eine Größe von einem Byte, weswegen wir dies auch hier in unseren Ausführungen annehmen wollen. Auch wenn Ihr Computer größere Speicherwörter benutzen sollte, dann gelten die grundlegenden Darlegungen dieses Kapitels dennoch. Wenn wir eine Variable deklarieren (vergleiche (6-5)), dann werden für den aufzunehmenden Wert dieser Variablen in Ihrem Wörterbucheintrag zwei Bytes reserviert. Wir können dort jetzt eine einfach genaue

Integer speichern. Wenn es uns nun gelänge, für diese Variable einen Speicherplatz von 100 Byte zu reservieren, dann könnte man 50 einfach genaue Integers unter diesem Variablennamen ablegen. Wir zeigen Ihnen jetzt, wie man den reservierten Speicherplatz für einen Variableneintrag im Wörterbuch vergrößern kann und wie man eine große Anzahl von Variablenwerten in diesen Speicherplatz schreiben bzw. aus diesem Speicherplatz holen kann.

ALLOT - Das FORTH-Wort **ALLOT** holt sich die oberste einfach genaue Integer vom Stack und erweitert den Speicherplatz des zuletzt definierten Wortes um die entsprechende Anzahl von Bytes. Die Anzahl von Bytes, um die diese Variable erweitert wird, entspricht also der Integer, die **ALLOT** vom Stack holt. Betrachten Sie dazu diese Kommandofolge:

```
VARIABLE KEE                                        (6-15a)
40 ALLOT                                            (6-15b)
```

Nach Ausführung von 6-15a enthält der Wörterbucheintrag für die Variable **KEE** zwei Byte, die zur Aufnahme der Variablenwerte bestimmt sind. Nach Ausführung von (6-15b) haben wir jedoch insgesamt 42 Byte an Speicherplatz unter dem Variablennamen **KEE** zur Verfügung. Zwei Byte wurden bereits bei Deklaration der Variablen (6-15a) zugewiesen, während das FORTH-Wort **ALLOT** in (6-15b) dafür sorgt, daß 40 weitere Byte hinzukommen. Die Stack-Relation für **ALLOT** lautet einfach:

```
n -->                                               (6-16)
```

Jetzt wollen wir einmal sehen, wie man einzelne (einfach genaue) Integers an dem so zugewiesenen Platz speichert bzw. von dort liest. Bekanntermaßen wird eine einfach genaue Integer ja in zwei Bytes (zwei Speicherwörtern) abgelegt. Wenn wir den Variablennamen **KEE** im FORTH-System eingeben, führt dies dazu, daß die Adresse dieser Variablen auf den Stack gelegt wird. Genauer gesagt: Die Erwähnung eines Variablennamens legt die Adresse des ersten der beiden Bytes auf den Stack, die zur Speicherung des Variablenwerts benötigt werden. Wenn die Kommandos ! oder @ aufgerufen werden, dann "wissen" Sie, daß die Ihnen zur Verfügung gestellte Adresse sich auf das erste Byte des Variablenwerts bezieht. Nach

Ausführung von (6-15b) haben wir unter dem Variablennamen **KEE** jedoch wesentlich mehr Platz zur Verfügung und können so mehr als nur eine einzige Integer speichern. Angenommen, wir wollen eine zweite einfach genaue Integer speichern, diesmal jedoch im dritten und vierten Speicherwort von **KEE.** Dies erreichen wir mit der folgenden Wortfolge:

```
98 KEE 2 + ! (RETURN)                                      (6-17)
```

Damit wird die einfach genaue Integer 98 im dritten und vierten Byte der Variable **KEE** abgelegt. Dies geht im einzelnen so: Zuerst legen wir den zu speichernden Wert (98) auf den Stack. Anschließend schreiben wir den Variablennamen **KEE** hin, wodurch die Adresse des ersten Speicherwortes der Variablen auf den Stack gelegt wird. Durch Eingeben von 2 + addieren wir nun 2 auf diese Adresse. Damit haben wir die Adresse des dritten Bytes, das zu **KEE** gehört. Deswegen speichert **!** den Wert 98 auch an dieser Stelle. Um den Variablenwert wieder auszulesen, bedienen wir uns eines ähnlichen Verfahrens:

```
KEE 2 + @                                                 (6-18)
```

Damit legen wir den Wert 98 auf den Stack. Denken Sie daran, daß man sich einen Variablenwert mittels @ holen kann, ohne daß dieser dadurch verändert wird. Wenn man eine Variable so erweitert, daß sie zur Speicherung einer ganzen Liste von Werten benutzt werden kann, dann bezeichnet man sie auch als <u>Array</u>. Lassen Sie uns nun unser erstes Beispielprogramm mit Arrays schreiben. Wieder machen wir es uns zur Aufgabe, den Teilnehmern an einem Kurs Noten zuzuweisen, und zwar auf der Basis ihres Leistungsdurchschnitts in den einzelnen Tests. Wir gehen davon aus, daß die Durchschnittswerte der Kursteilnehmer bereits in einem Array mit dem Namen **AVERAGE** gespeichert sind. Das erste Element dieses Arrays fällt etwas aus der Reihe. Es gibt nämlich die Anzahl der Kursteilnehmer an und stellt keinen Durchschnittswert dar. Alle anderen einfach genauen Integers sind jedoch Punktzahlen, die den Leistungsdurchschnitt der einzelnen Studenten darstellen. Weiterhin gehen wir davon aus, daß die Position eines Durchschnittswerts im Array mit der Klassennummer des Teilnehmers übereinstimmt. Der Teilnehmer Nummer 1 hat seine Daten also an erster

Postition, Teilnehmer Nummer 2 an zweiter usw. Dies können wir machen, da bei der Arbeit mit Arrays in der Regel mit dem Index 0 begonnen wird, d.h., der erste Dateneintrag im Array befindet sich an Position 0. Position 0 ist in unserem Beispiel aber die Anzahl der Teilnehmer, während Position 1 der Durchschnitt des ersten Teilnehmers, Position 2 der des zweiten Teilnehmers ist usw.

```
 0 (Beispiel mit Arrays und ALLOT)
 1 VARIABLE DURCH    200 ALLOT
 2 : BEWERTUNG  DURCH  @  1 +   1
 3       DO CR 2 I *  DURCH   +  @    59    -    0>    IF
 4          ." STUDENT NO. "    I   .    ." BESTEHT "
 5           ELSE  ."STUDENT NO.   "    I   ." FAELLT DURCH "
 6           THEN
 7       LOOP  ;
 8
 9 (Beispiel fuer Dateneingabe in Arrays)
10 : INPDRCH  CR   ." Bitte Teilnehmerzahl eingeben "
11          #IN   DUP   DURCH !    1   +  1   CR   DO
12            ." Durchschnitt des Teilnehmers Nr. "  I .   #IN
13            CR  DURCH  I  2  *   +    !
14          LOOP  CR    ;
15
```

ABBILDUNG 6-3: Ein FORTH-Programm, das mit Arrays arbeitet

Unser Programm (vgl. Abb. 6-3) gibt die Nummer jedes Teilnehmers aus und druckt dahinter BESTEHT oder FäLLT DURCH. Falls der Punktedurchschnitt den Wert 60 übersteigt, dann besteht der Teilnehmer. Jetzt zu den Einzelheiten dieses Programms. Als erstes reservieren wir uns in Zeile 1 eine Variable mit dem Namen **DURCH**, aus der wir anschließend einen Array machen, indem wir 200 **ALLOT** eingeben. Dadurch kommen 200 weitere Byte zum Speicherplatz von **DURCH** hinzu. Dies bedeutet, daß weitere 100 einfach genaue Integers gespeichert werden können. Insgesamt kann der Array **DURCH** also 101 sog. Elemente aufnehmen. Im Rest der Abbildung 6-3 werden zwei neue FORTH-Wörter definiert. Das Wort **BEWERTUNG** berechnet das Abschlußergebnis und gibt es aus. Ehe dieses Wort angewendet werden kann, müssen jedoch die Werte in den Array **DURCH** gebracht werden; dazu dient das Wort **INPDRCH.** Wir nehmen einmal an, daß **DURCH** bereits mit Werten versorgt ist und sehen uns an,

was **BEWERTUNG** nun macht. Die erste Zahl im Array **DURCH** stellt die Anzahl der Kursteilnehmer dar. Wir gehen davon aus, daß der Kurs weniger als 100 Teilnehmer besitzt, da wir ja nur Platz für insgesamt 100 Elemente mittels **ALLOT** reserviert haben. Achten Sie genau darauf, daß Sie nicht mehr Speicherplatz verwenden, als Sie reserviert haben. Sollten Sie in der vorliegenden Situation etwa versuchen, das 102te Element von **DURCH** zu beschreiben, dann können Sie unter Umständen wertvolle Information im FORTH-Wörterbuch dadurch zerstören.

Jetzt zu dem Wort **BEWERTUNG.** In Zeile 2 besorgen wir uns die Adresse des Arrays, indem wir seinen Namen **DURCH** hinschreiben. Genaugenommen steht jetzt die Adresse des ersten Datenelements auf dem Stack. Dieses Datenelement besorgen wir uns nun mit @. Vereinbarungsgemäß haben wir nun an oberster Stack-Position die Anzahl der Kursteilnehmer stehen. Zu dieser Zahl addieren wir nun 1 hinzu, pushen eine weitere 1 auf den Stack und haben damit die Parameter für eine DO-Schleife eingerichtet. In diese Schleife treten wir in Zeile 3 mit dem bekannten Schlüsselwort **DO** ein. Das Wort entfernt den Anfangswert für den Schleifenindex sowie den Testwert vom Parameter-Stack und legt ihn auf den Return-Stack. Der erste Befehl in der Schleife - **CR** - sorgt dafür, daß die Programmausgaben lesbarer werden. Jetzt multiplizieren wir den Schleifenindex mit 2 und holen die Anfangsadresse des Arrays **DURCH**, auf die wir den soeben erhaltenen Wert (Schleifenindex mal 2) addieren. Zu diesem Zeitpunkt befindet sich somit die Originaladresse unserer Daten, erhöht um zweimal den Schleifenindex auf dem Stack. Als nächstes führen wir das Wort @ aus. Dies bedeutet, daß bei jedem Schleifendurchgang der Durchschnittswert eines Kursteilnehmers aus dem Arbeitsspeicher geholt und auf den Stack gelegt wird. Wir können nicht einfach den Schleifenindex benutzen, um auf die einzelnen Elemente des Arrays **DURCH** zuzugreifen, da ja zur Speicherung einer einfach genauen Integer zwei Byte benötigt werden. Das ist der Grund, warum wir vor dem Zugriff auf die Daten den Schleifenindex mit 2 multiplizieren.

Jetzt können wir testen, ob der Teilnehmer bestanden hat: wir legen 59 auf den Stack und subtrahieren diesen Wert vom Durchschnittswert des Kursteilnehmers. Wenn das Ergebnis größer Null ist, dann besteht der Teilnehmer, ist die Differenz jedoch kleiner Null, dann ist er durchgefallen. Diese beiden Fälle unterscheiden wir, indem wir uns von dem Vergleichswort **0>** ein Flag auf den Stack legen lassen. Falls dieses Flag "wahr" ist, kommt der Teilnehmer durch, ansonsten fällt er durch. Mittels **IF-ELSE-**

THEN geben wir die jeweils passende Meldung aus und vergessen nicht, die laufende Nummer des Teilnehmers dabei mit aufzunehmen.

Wir könnten die Werte in den Array **DURCH** bringen, indem wir das im Beispiel (6-17) vorgestellte Verfahren benutzen. Dies ist jedoch wesentlich mühseliger als das selbstdefinierte Wort **INPDRCH**, das Sie auch in Abbildung 6-3 sehen. Das Wort fordert den Benutzer selbsttätig zur Eingabe der richtigen Daten auf. Er muß nur mehr die Durchschnittswerte eingeben und braucht sich nicht mehr darum zu kümmern, daß diese an der richtigen Stelle im Array abgelegt werden. Im einzelnen geht das folgendermaßen. In Zeile 10 der Abbildung 6-3 geben wir die Meldung BITTE ANZAHL DER TEILNEHMER EINGEBEN:. Diese Teilnehmerzahl lesen wir dann in Zeile 11 mit dem FORTH-Wort **#IN** ein. Der Benutzer sieht also ein Fragezeichen auf dem Bildschirm, und das Programm hält an, bis er die gewünschte Zahl eingegeben hat. Diese Zahl wird als nächstes dupliziert und an Position Null des Arrays **DURCH** gespeichert. das erreichen wir ganz einfach dadurch, daß wir auf den Variablennamen den Speicherbefehl **!** folgen lassen. Der Rest von Zeile 11 bereitet die Schleifenparameter vor. Vor Eintritt in die Schleife sorgen wir noch mittels **CR** dafür, daß die nachfolgende Eingabeaufforderung auf einer eigenen Zeile zu stehen kommt. In der Zeile 12 geben wir die Meldung: "Durchschnitt des Teilnehmers Nr." aus. Hinter dieser Meldung zeigen wir dem Bediener den Schleifenindex. Das Eingabewort **#IN** besorgt sich nun vom Benutzer die fragliche Punktzahl. Diese wird in Zeile 13 an der richtigen Stelle abgespeichert. Dazu pushen wir die Anfangsadresse von **DURCH** auf den Stack und addieren dazu den mit 2 multiplizierten Schleifenindex. Das Vorgehen ist hier also völlig analog zu dem, das wir in **BEWERTUNG** bereits kennengelernt haben. Nach einem Durchgang durch die Schleife ist im richtigen Element von **DURCH** der Punktedurchschnitt des Teilnehmers abgespeichert.

ARRAY - Wie man Arrays mit **ALLOT** vereinbart, haben wir bereits erfahren. Im MMSFORTH und anderen FORTH-Systemen gibt es jedoch ein bequemeres Verfahren, mit dem Arrays vereinbart werden können. Dazu benutzt man das Wort **ARRAY** in folgender Weise:

```
100 ARRAY DURCH                                    (6-19)
```

Diese Kommandofolge bewirkt das gleiche wie die Zeile 1 von Abbildung 6-3. Es wird also ein Wörterbucheintrag für die Varia-

ble **DURCH** eingerichtet, der außerdem Platz für 101 einfach genaue Integers enthält. Wie wir jetzt wissen, werden dazu 202 Byte im Arbeitsspeicher des Computers reserviert. Bei der Arbeit mit **ALLOT** muß man die Anzahl der Bytes angeben, die reserviert werden sollen, während **ARRAY** direkt mit Array-Elementen arbeitet. Im Falle von 6-19 werden diese Elemente fortlaufend von 0 bis 100 durchnumeriert. Das Wort **ARRAY** hat folgende Stack-Relation:

```
n -->                                                    (6-20)
```

Ein mit **ARRAY** vereinbarter Array kann insgesamt n + 1 Elemente aufnehmen, die von 0 bis n durchnumeriert sind. Der Platzbedarf für diesen Array beträgt 2(n+1) Byte. Um die Adresse eines Array-Elements auf den Stack zu bekommen, geben wir lediglich die Elementnummer und den Namen des Arrays in dieser Reihenfolge ein. Durch Ausführung von

```
3 DURCH                                                  (6-21)
```

erhalten wir so z.B. die Adresse des ersten Bytes des vierten Elements von **DURCH** auf dem Stack. Rufen Sie sich noch einmal ins Gedächtnis, daß das erste Element die Nummer 0 besitzt. Deshalb erreichen wir mit (6-21) auch das vierte Array-Element.

Als Beispiel für den Einsatz von **ARRAY** haben wir das Programm aus der Abbildung 6-3 noch einmal neu geschrieben. Sie sehen die geänderte Fassung in Abbildung 6-4. Im wesentlichen handelt es sich dabei um das gleiche Programm wie in Abbildung 6-3, außer daß wir den Array **DURCH** mit dem FORTH-Befehl **ARRAY** vereinbaren. Außerdem sind jetzt die Adreßberechnungen einfacher, da wir den Schleifenindex nicht mehr mit 2 multiplizieren müssen. Werfen Sie noch einen Blick auf Zeile 2: Dort besorgen wir uns mittels 0 **DURCH** die Adresse des ersten Elements des Arrays.

Wie Sie sehen können, benutzt man zur Adressierung von Arrays, die mit **ALLOT** eingerichtet wurden, ein anderes Verfahren als für solche, die Sie sich mittels **ARRAY** besorgt haben. Verwechseln Sie nicht diese beiden Methoden! Das Verfahren mit **ARRAY** ist zweifelsohne einfacher, leider aber nicht Bestandteil von FORTH-79. Sollte Ihr System diese Möglichkeit zur Verfügung stellen, so

kann es sein, daß Sie zuvor die betreffenden Blöcke in den Speicher laden müssen, ehe Sie auf diese Möglichkeit zurückgreifen können.

```
 0 ( Ein weiteres Beispiel fuer Arrays )
 1 100 ARRAY DURCH
 2 : BEWERTUNG  0 DURCH @  1 +  1
 3      DO CR  I DURCH @  59   -   0>   IF
 4         ." STUDENT NO. "  I     ." BESTEHT "
 5          ELSE ." STUDENT NO. "  I  ." FAELLT DURCH "
 6          THEN
 7      LOOP ;
 8
 9 (Dateneingabe in einen Array)
10 : INPDRCH  CR ." Bitte Anzahl der Teilnehmer eingeben: "
11       #IN  DUP  0 DURCH !  1 +  1  CR  DO
12         ." Durchschnittswert des Studenten Nr. "  1  .  #IN
13         CR  I  DURCH  !
14       LOOP  CR ;
15
```

ABBILDUNG 6-4: Beispiel für den Einsatz von **ARRAY**

6.3.1 Arrays und doppelt genaue Integers

Sie können auch doppelt genaue Integers in Arrays speichern. Auch dazu gibt es wieder zwei Verfahren, wobei wir uns zuerst dem mittels **ALLOT** zuwenden wollen. Wenn wir mit diesem FORTH-Wort den Speicherplatz einer Arrayvariablen erweiterten, dann haben wir bisher immer für jede zu speichernde einfach genaue Integer zwei Byte reserviert. Da doppelt genaue Integer einen doppelt so grosen Speicherplatzbedarf haben, müssen wir entsprechend 4 Byte für jeden zu speichernden Wert vereinbaren:

```
2VARIABLE WORK 40 ALLOT                                    (6-22)
```

Wir richten also zuerst eine doppelt genaue Variable mit dem Namen **WORK** ein und erweitern deren Speicherkapazität durch 40

ALLOT um weitere 40 Byte. Wenn wir jetzt **WORK 4 +** eingeben, dann haben wir danach die Adresse der zweiten doppelt genauen Integer in dem Array **WORK** auf dem Stack. Wenn wir doppelt genaue Arrays in einer Schleife verarbeiten und über den Schleifenindex ansprechen wollen, dann müssen wir den Schleifenindex mit 4 multiplizieren. Dies ist analog zu dem Verfahren in Abbildung 6-3, in dem wir den Schleifenindex mit 2 multipliziert haben. Zum Speichern und Lesen von doppelt genauen Array-Elementen müssen Sie außerdem die Wörter **2!** und **2@** verwenden.

Es ist nicht einmal notwendig, doppelte Genauigkeit für die Ausgangsvariable zu vereinbaren. Wir hätten in (6-22) **WORK** genausogut mit **VARIABLE** vereinbaren können. In diesem Fall kann man in den ersten beiden Bytes des Arrays eine einfach genaue Integer speichern, während die nachfolgenden Bytes jeweils in Vierergruppen zur Speicherung von doppelt genauen Werten dienen. Allerdings macht dieses Verfahren die Adreßberechnung für die Array-Elemente umständlicher.

Wenn Ihr FORTH-System über diese Möglichkeit verfügt, dann können Sie auch mit dem Befehl **DARRAY** Arrays mit doppelt genauen Integers vereinbaren. Auch dieses Wort ist kein Bestandteil von FORTH-79, aber im MMSFORTH zu finden. Es funktioniert im wesentlichen genauso wie **ARRAY.** Seine Stack-Relation lautet:

```
d -->                                                    (6-23)
```

Nach Ausführung von

```
100 DARRAY WORK                                          (6-24)
```

haben wir also einen Array mit dem Namen **WORK** eingerichtet, der Platz für 101 doppelt genaue Integers hat, insgesamt also 404 Byte im Arbeitsspeicher beansprucht. Wenn wir das vierte Element dieses Arrays auf den Stack legen wollen, dann geben wir ein

```
4 WORK 2@                                                (6-25)
```

6.3.2 Arrays und Gleitkommazahlen

Gleitkommazahlen sind kein Bestandteil von FORTH-79, können aber in MMSFORTH und ähnlichen Systemen benutzt werden. Die hier dargestellten Array-Wörter sind die von MMSFORTH. Sie sind also nicht standardisiert. Zur Einrichtung eines Arrays mit einfach genauen Gleitkommazahlen dient **2ARRAY**, das im wesentlichen genauso wie **ARRAY** funktioniert, außer daß eben mit einfach genauen Gleitkommazahlen gearbeitet wird. Ähnlich kann man mit **4ARRAY** Arrays mit doppelt genauen Gleitkommazahlen einrichten.

6.3.3 Arrays mit Konstanten

Gelegentlich ist es sehr nützlich, über einen Array zu verfügen, der eine Folge von Konstanten enthält. Dies kann in FORTH-79 leicht erreicht werden, indem man die Wörter **CREATE** und **,** einsetzt. Man kann diese beiden Wörter zwar getrennt für sich verwenden, davon wird dem Anfänger jedoch abgeraten. Wenn Sie nicht genau mit der Arbeitsweise Ihres FORTH-Systems vertraut sind, dann sollten Sie sie also nur zusammen einsetzen. Bei Ausführung von **CREATE** wird ein Wörterbucheintrag für den Namen eingerichtet, der auf das Schlüsselwort folgt. So sorgt also

```
CREATE TAB                                        (6-26)
```

dafür, daß für das Wort **TAB** ein Wörterbucheintrag eingerichtet wird. Allerdings wird zu diesem Zeitpunkt kein Speicherplatz für Daten vereinbart! Mit dem FORTH-Kommando **,** wird die oberste Integer vom Stack entfernt und dafür im nächsten freien Wörterbucheintrag ein Speicherbereich reserviert. Anschließend wird in diesem Speicherbereich die Integer vom Stack abgelegt. Angenommen, wir wollen in unserem Konstanten-Array **TAB** sechs Werte speichern. Dies erreichen wir durch folgende Kommandos:

```
CREATE TAB 15 , 20 , 35 , 40 , 50 , 60 ,          (6-27)
```

Die Zahlen und Kommas sind jeweils durch Leerzeichen getrennt, damit sie von FORTH als eigene Wörter erkannt werden. Wenn wir auf das FORTH-Kommando **CREATE** also den Namen des Arrays folgen lassen und dahinter die entsprechenden Werte durch Komma getrennt aufführen, dann werden diese Werte im Wörterbuch von FORTH an den passenden Stellen abgelegt. Beispiel (6-27) sorgt dafür, daß im Wörterbuch ein Eintrag für **TAB** vorgesehen wird. Als nächstes kommt die 15 auf den Stack. Das nächste Wort - , - reserviert 2 Byte direkt hinter dem Wörterbucheintrag für **TAB** und speichert dort die 15. Entsprechend werden die anderen aufgeführten Konstantenwerte gespeichert.

Bei einer nachfolgenden Ausführung des Wortes **TAB** findet sich an oberster Stack-Position die Adresse des ersten Daten-Bytes (in diesem Fall der Zahl 15). Nocheinmal: **CREATE** reserviert keinen Speicherplatz für die Daten im Wörterbuch. Ist der Array **TAB** mittels (6-17) erst einmal eingerichtet, dann kann man ihn wie jeden anderen Array mit einfach genauen Integers behandeln. Man kann sogar Werte in **TAB** speichern, indem man sich der bereits vorgestellten Verfahren bedient. Wenn wir einen Array nicht mit Konstantenwerten initialisieren müssen, dann sollte nicht **CREATE** benutzt werden, da die Arbeit mit Arrays sonst zu umständlich wird. In solchen Fällen sollten Sie den normalen Weg gehen und mit **ALLOT** arbeiten.

6.4 Mehrdimensionale Arrays

Manche Anwendungen erfordern die Verarbeitung mehrerer Datenlisten (Arrays), die miteinander verknüpft sind. So könnte in dem einen Array z.B. die Prüfungsleistung von Studenten enthalten sein, während der andere Array die Matrikelnummern enthält. Es wäre nun wünschenswert, eine Verbindung zwischen der Leistung des Studenten und seiner Matrikelnummer herzustellen. Für dieses Problem bieten mehrdimensionale Arrays eine elegante Lösung. Sehen wir uns ein Beispiel an. Wir haben für die Teilnehmer an einem Kurs eine Matrikelnummer sowie die Punktzahl in zwei Tests. Wir wollen den Durchschnitt des Studenten berechnen und neben seiner Matrikelnummer ausgeben. Dies macht das FORTH-Programm in Abbildung 6-5. Die ersten drei Zeilen dienen zur Vereinbarung von drei unterschiedlichen Arrays, die jedoch allesamt gleich lang sind.

```
 0 ( Beispiel fuer parallelverarbeitung mehrerer Arrays )
 1 VARIABLE  IDNUMB   200    ALLOT
 2 VARIABLE  GRADE1   200  ALLOT
 3 VARIABLE  GRADE2   200  ALLOT
 4 : CLASS        IDNUMB @      1 +   1   DO
 5       CR 2  I *  GRADE1  + @  2  I *  GRADE2  +  @  +
 6         ." STUDENT NO.  "    I  2  *   IDNUMB    +  @  .
 7          ." HAT DEN DURCHSCHNITT   "  2  /   .
 8       LOOP    ;
 9 (Beispiel fuer Werteingabe in Arrays)
10 : GRADEIN  CR   ." BITTE TEILNEHMERZAHL EINGEBEN "
11            #IN   DUP   IDNUMB   !    1  +  1    CR    DO
12            CR   ." MATRIKELNUMMER UND BEWERTUNG EINGEBEN "
13                 #IN  #IN  #IN
14           I 2 *  GRADE2 + !  I 2 * GRADE1 + ! I 2 * IDNUMB +
15       LOOP ;
```

ABBILDUNG 6-5: Ein Beispiel für mehrfache Arrays

Im Array **IDNUMB** speichern wir die Matrikelnummern des Studenten. **GRADE1** enthält die Punktzahl im ersten Test, während **GRADE2** die Ergebnisse des zweiten Tests beinhaltet. Zusätzlich merken wir uns in Position 0 von **IDNUMB** die Anzahl der Teilnehmer im Kurs. Die entsprechende Position der beiden anderen Arrays bleibt indessen ungenutzt.

Wir gehen jetzt davon aus, daß diese drei Arrays mit Daten versorgt sind und wenden uns der Arbeitsweise des FORTH-Wortes **CLASS** zu. (Vgl. Abb. 6-5). Die erste Zeile des Programms erhöht den "Teilnehmerzähler" um 1; dazu besorgen wir uns die Anfangsadresse des Arrays, indem wir seinen Namen rufen (**IDNUMB**) und uns anschließend mittels @ das nullte Datenelement dieses Arrays auf den Stack legen lassen. Auf diesen Wert (die Anzahl der Teilnehmer) addieren wir als nächstes 1 mit der Kommandofolge **1 +**. Die Anzahl der Teilnehmer soll nämlich als Testwert einer DO-Schleife dienen, wobei wir bereits wissen, daß der Testwert in einer solchen DO-Schleife um 1 höher sein muß als die Anzahl der gewünschten Durchläufe (vorausgesetzt, der Index beginnt mit 1, wie es hier der Fall ist). Das letzte Wort auf Zeile 4 leitet die Schleife ein, welche jetzt genausooft durchlaufen wird, wie es Teilnehmer im Kurs gibt. Die Zeile 5 der Abbildung 6-5 dient dazu, für den ersten Studenten die Ergebnisse in den beiden Tests zu ermitteln. Dabei gehen wir wie üblich vor: Der Schleifenindex

wird mit 2 multipliziert und anschließend diese Zahl auf die Startadresse des Arrays addiert, den wir bearbeiten wollen. Dies machen wir zweimal, einmal für **GRADE1** und einmaL für **GRADE2.** Danach haben wir die Punktzahl in den beiden Tests auf dem Stack und können sie mittels + addieren, um für die Durchschnittsberechnung alles vorzubereiten. Zeile 6 dient dazu, neben einer Textmeldung die Matrikelnummer des Studenten auszugeben; die Matrikelnummer erreichen wir wiederum, indem wir den Laufindex mit 2 multiplizieren, diese Zahl auf die Startadresse des Arrays addieren und das Datenelement an der so berechneten Position mittels @ holen. In Zeile 7 wird nun der Durchschnitt berechnet und ausgegeben. Nach Abarbeitung von Zeile 6 ist die Summe der Punkte in den beiden Tests oberster Stack-Eintrag. Diese Zahl teilen wir durch 2 und geben den Quotienten aus. Damit ist die Schleife beendet, und das **LOOP** auf Zeile 8 sorgt für einen Wiedereintritt, so lange die Testbedingung noch nicht erfüllt ist.

In den Zeilen 10 bis 15 der Abbildung 6-5 sehen Sie ein Programm zur Eingabe der Daten. Dies folgt weitestgehend dem Verfahren, das wir aus Abbildung 6-3 schon kennen, außer daß nun Daten in drei Arrays eingegeben werden müssen. Die Daten werden in folgender Reihenfolge auf den Stack gelegt: Matrikelnummer, Test1 und Test2. Wir lesen sie also in umgekehrter Reihenfolge vom Stack.

Dieses Programm kann auch mit doppelt genauen Integers geschrieben werden; der Algorithmus bleibt der gleiche. Sie müssen nur die passenden Wörter für die Manipulation von doppelt genauen Integers einsetzen.

6.4.1 Zweidimensionale Arrays

Im letzten Beispiel haben wir drei Arrays im Programm verwendet, die inhaltlich aufeinander bezogen sind. Jeder dieser Arrays war zur Aufnahme einer anderen Art von Daten bestimmt. In solchen Fällen, in denen mehrere aufeinander bezogene Datensätze gegeben sind, ist es oft bequemer, mit zweidimensionalen Arrays zu arbeiten. Alle bisher betrachteten Arrays waren eindimensional. Einen eindimensionalen Array kann man sich als eine (beliebig lange) Liste von Daten vorstellen, oder anders ausgedrückt, als eine Datenzeile mit (beliebig vielen) Datenspalten. Im Unterschied dazu kann ein zweidimensionaler Array nicht nur beliebig viele

Datenspalten, sondern auch eine größere Anzahl Datenzeilen aufweisen. Zweidimensionale Arrays haben daher die logische Struktur einer Wertetabelle; wer sich noch nichts unter einem zweidimensionalen Array vorstellen kann, der denke an die bekannten Entfernungstabellen, in denen die Entfernung zwischen Städten in km abzulesen ist. Hierbei handelt es sich um ein Beispiel für einen zweidimensionalen Array.

Natürlich können wir die Daten für das Beispielprogramm in Abbildung 2-5 auch in einem zweidimensionalen Array speichern. Jede Zeile dieses Arrays entspricht dann einem Studenten, wobei in Spalte 0 die Matrikelnummer, in Spalte 1 das Ergebnis des ersten, in Spalte 2 das Ergebnis des zweiten Tests zu finden ist (Beachten Sie, daß wir die Datenelemente mit 0 beginnend durchnumerieren!)

```
 0 ( Beispiel fuer einen zweidimensionalen Array )
 1 2 100 2ARRAY GRADE
 2 : CLASSAV    0 0 GRADE @  1 +  1  DO
 3          CR I 1 GRADE  @  I 2 GRADE @  +
 4          ." STUDENT NO. "  I 0 GRADE @  .
 5          ." HAT DEN DURCHSCHNITT "  2 /  .
 6     LOOP   ;
 7
 8 (Werteversorgung des Arrays ueber Tastatur)
 9 : GRADEIN  CR  ." BITTE ANZAHL DER TEILNEHMER EINGEBEN "
10            #IN  DUP  0  0  GRADE  !  1 + 1  DO
11            CR  ." MATRIKELNUMMER UND BEWERTUNGEN EINGEBEN "
12                 #IN  #IN  #IN
13            I 2 GRADE  !  I 1 GRADE   !  I 0 GRADE   !
14     LOOP  ;
15
```

ABBILDUNG 6-6: Ein Beispiel für zweidimensionale Arrays

Der Befehl zur Definition eines zweidimensionalen Arrays ist kein Bestandteil von FORTH-79, findet sich jedoch in MMS-FORTH. Er lautet **2ARRAY** und wird folgendermaßen angewendet:

n_1 n_2 2ARRAY GRADE (6-28)

Das Wort **2ARRAY** erwartet zwei einfach genaue Integers auf dem Stack, die es davon entfernt. Es vereinbart daraufhin einen zweidimensionalen Array - in diesem Fall mit dem Namen **GRADES** -, der n_1 + 1 Zeilen und n_2 +1 Spalten hat. Die erste Zeile und Spalte tragen jeweils die Nummer 0. Die Stack-Relation **2ARRAY**:

n_1 n_2 --> (6-29)

Die Werte für n_1 und n_2 müssen sich dabei zwischen 1 und 254 bewegen.

Wenden wir uns jetzt der Frage zu, wie man Daten in einem zweidimensionalem Array speichern bzw. aus ihm auslesen kann. Um beispielsweise die Adresse des Datenelements auf den Stack zu bekommen, das sich in Zeile n_1 und Spalte n_2 des Arrays **GRADE** befindet, müssen wir eingeben:

n_1 n_2 GRADE (6-30)

Das bedeutet, daß wir die einfach genaue Integer, die sich in Zeile 4, Spalte 3 des Arrays **GRADE** befindet, mit folgender Eingabe erreichen:

4 3 GRADE

Als Beispiel für den Einsatz zweidimensionaler Arrays wollen wir das Programm aus Abbildung 6-5 neu schreiben. Sie sehen die veränderte Version in der Abbildung 6-6.

Wir brauchen jetzt nur mehr eine Zeile zur Einrichtung des nötigen Arrays, der diesmal jedoch zwei Dimensionen hat; das besorgt die Zeile 1 des Beispielprogramms. Nach ihrer Ausführung ist im FORTH-System Speicherplatz für einen zweidimensionalen Array mit dem Namen **GRADE** reserviert, der 101 Zeilen mit jeweils 3 Spalten besitzt. Wir gehen davon aus, daß die Gesamtzahl der Teilnehmer am Kurs im Element (0,0) des Arrays gespeichert ist. Sie darf 100 nicht überschreiten, da sonst die Arraygrenzen gesprengt werden. Übrigens bezieht man sich auf Elemente eines Arrays durch Angaben in der Form (Zeile, Spalte). In Zeile 2 beginnt die eigentliche Definition des neuen Wortes **CLASSAV**; als erstes werden die Para-

meter für eine DO-Schleife eingerichtet, wozu wir uns die Anzahl der Teilnehmer besorgen und um 1 erhöhen. Das Verfahren entspricht also genau dem von Abbildung 6-5, wir müssen jetzt jedoch zwei Werte auf den Stack legen, um an das erste Datenelement des Arrays **GRADE** zu gelangen. In Zeile 3 holen wir die Datenelemente aus **GRADE**, die in der dem Schleifenindex entsprechenden Zeile und den Spalten 1 und 2 zu finden sind. Die Matrikelnummer des Teilnehmers finden wir in Spalte 0. Wir besorgen sie uns in Zeile 4, d.h., wir holen das Element aus dem Array, das in der dem Schleifenindex entsprechenden Zeile und Spalte 0 zu finden ist. Der Rest des Programms entspricht genau dem aus Abbildung 6-5.

Die Zeilen 8 bis 13 der Abbildung 6-6 zeigen die Definition des FORTH-Wortes **GRADEIN**, mit dem der Benutzer Daten in den Array **GRADE** eingeben kann. Auch dieses Programm folgt in seinen Verarbeitungsschritten dem aus Abbildung 6-5, außer daß jetzt die nötigen Verfahren zur Speicherung in zweidimensionalen Arrays angewendet werden. Beachten Sie, daß das Programm aus Abbildung 6-6 einfacher ist als das in Abbildung 6-5.

6.4.2 Zweidimensionale Arrays und doppelte Genauigkeit

Mit dem FORTH-Kommando **2DARRAY** kann man einen zweidimensionalen Array vereinbaren, in dem doppelt genaue Integers gespeichert werden. Das Wort ist zwar kein Bestandteil von FORTH-79, wurde jedoch in MMSFORTH aufgenommen. Die Details des Umgangs mit **2DARRAY** entsprechen denen für **ARRAY**. Die Stack-Relation lautet:

$$n_1 \; n_2 \; \text{-->} \qquad (6\text{-}31)$$

Die Integers, mit denen der Umfang des Arrays festgelegt wird, müssen einfach genau sein, sich also zwischen 0 und 65535 bewegen. Zum Speichern und Wiederauffinden von Daten in solchen Arrays müssen die Kommandos **2@** und **2!** verwendet werden.

6.4.3 Zweidimensionale Arrays und Gleitkommazahlen

MMSFORTH und andere FORTH-Systeme stellen Wörter zur Verfügung, mit denen zweidimensionale Arrays von Gleitkommazahlen definiert werden können. Da sie nicht zum FORTH-79-Standard gehören, können sich diese Wörter von System zu System unterscheiden. Für einfach genaue Gleitkommazahlen definiert man einen zweidimensionalen Array mit dem Wort **22ARRAY**. Die Handhabung eines solchen Arrays entspricht der, die wir von **2ARRAY** schon kennen, außer daß jetzt keine einfach genauen Integers, sondern Gleitkommazahlen im Array gespeichert werden. Für den Zugriff (Lesen und Schreiben) nimmt man die Wörter **2@** und **2!**. Die Stack-Relation entspricht der von (6-29). Die Zahlen, die die Abmessungen des Arrays festlegen (n_1 und n_2) sind einfach genaue Integers. Auch die Zahlen, mit deren Hilfe man auf ein einzelnes Element des Arrays zugreift, müssen einfach genaue Integers sein.

Fernerhin ist es möglich, durch das FORTH-Wort **24ARRAY** eine zweidimensionalen Array für doppelt genaue Gleitkommazahlen zu vereinbaren. Bis auf den Umstand, daß für den Datenzugriff die Wörter 4@ und **4!** verwendet werden müssen, gleicht die Arbeit mit solchen Arrays denen mit Gleitkommazahlen von einfacher Genauigkeit.

6.5 Übungsaufgaben

In einigen der folgenden Übungsaufgaben sollen Sie neue FORTH-Wörter definieren bzw. Programme schreiben. Überprüfen Sie diese, indem Sie sie auf Ihrem Computer laufen lassen. Versuchen Sie, möglichst kurze Programme zu schreiben, indem Sie Teilaufgaben durch Modularisierung an Unterwörter vergeben.

6-1 Erörtern Sie den Unterschied zwischen Konstanten und Variablen.

6-2 Schreiben Sie ein Programm, das den Umfang und den Flächeninhalt eines Kreises berechnet und dabei die Kreiszahl π als Konstante verwendet. Das Programm erwartet den Radius des Kreises auf dem Stack. Zur Berechnung des Kreisumfangs dient die Formel $2\pi r$, während die Formel für die Berechnung der

Kreisfläche $r^2\pi$ lautet. Legen Sie in diesem Programm für den Wert 3,142 zugrunde. Falls nötig, sollten Sie Ihre Werte skalieren.

6-3 Erörtern Sie den Vorteil, der sich durch den Einsatz von Konstanten in FORTH-Wörtern ergibt.

6-4 Wiederholen Sie Aufgabe 6-2, wobei Sie diesmal für π den Wert 3,1415927 ansetzen.

6-5 Wiederholen Sie Aufgabe 4-8 mit Variablen.

6-6 Wiederholen Sie Aufgabe 4-10 mit Variablen.

6-7 Wiederholen Sie Aufgabe 4-16 mit Variablen. Die Punktzahlen sollten nicht auf den Stack gelegt, sondern direkt vom Benutzer im Programm eingegeben werden.

6-8 Schreiben Sie ein FORTH-Wort, das unter Verwendung von Variablen die Fakultät berechnet.

6-9 Wiederholen Sie Aufgabe 6-8 mit doppelt genauen Integers.

6-10 Wiederholen Sie Aufgabe 6-2 mit einer Gleitkommakonstante.

6-11 Wiederholen Sie Aufgabe 6-10, wobei Sie sowohl Gleitkommavariable als auch Gleitkommakonstanten einsetzen sollen.

6-12 Wiederholen Sie Aufgabe 6-11 mit doppelt genauen Variablen und Konstanten. Legen Sie den Wert von zugrunde, der in Aufgabe 6-4 angegeben ist.

6-13 Wiederholen Sie Aufgabe 6-8 mit Gleitkommavariablen.

6-14 Wiederholen Sie Aufgabe 6-13 mit doppelt genauen Gleitkommavariablen.

6-15 Was ist ein Array?

6-16 Wiederholen Sie Aufgabe 5-10, berechnen Sie diesmal jedoch die Provision einer größeren Anzahl von Verkäufern. Die Verkäufer werden über ihre Personalnummer identifiziert. Ihr Programm sollte Arrays verwenden. Gehen Sie davon aus, daß die Firma nicht mehr als 250 Verkäufer hat. In Ihrem Pro-

gramm sollte auch die Möglichkeit vorgesehen sein, auf einfache Art Daten in den Array einzugeben bzw. von dort zu lesen.

6-17 Wiederholen Sie Aufgabe 6-16 mit doppelt genauen Integers.

6-18 Schreiben Sie ein Programm, das pythagoreische Zahlentripel berechnet. Speichern Sie die Daten in einem Array. Das Programm sollte die im Array gespeicherten Werte ausgeben können.

6-19 Wiederholen Sie Aufgabe 6-18 mit doppelt genauen Integers.

6-20 Wiederholen Sie Aufgabe 6-16 mit einfach genauen Gleitkommazahlen.

6-21 Wiederholen Sie Aufgabe 6-16 mit doppelt genauen Gleitkommazahlen.

6-22 Wiederholen Sie Aufgabe 6-18 mit einfach genauen Gleitkommazahlen.

6-23 Wiederholen Sie Aufgabe 6-18 mit doppelt genauen Gleitkommazahlen.

7

Zeichen und Zeichenfolgen

7 Zeichen und Zeichenfolgen

Mit den FORTH-Wörtern, die wir bisher selbst definiert haben, konnten wir nur Zahlen manipulieren. Jetzt wollen wir sehen, wie man auch mit Buchstaben und Buchstabenfolgen (Zeichenketten oder Strings) in Programmen arbeiten kann. Dabei beschränkt sich unsere Darstellung nicht nur auf die Buchstaben des Alphabets, sondern auch auf Ziffernzeichen (0-9) und Interpunktionszeichen und andere Sonderzeichen. Für diese Art von Zeichen gibt es einen Fachausdruck: Man bezeichnet die Buchstaben, Ziffern und Sonderzeichen mit dem umfassenden Ausdruck alphanumerische Daten. Einzelne Zeichen bilden zusammengeschrieben sog. Zeichenketten oder Strings. Wir werden in diesem Kapitel auch FORTH-Wörter kennenlernen, mit denen man Strings manipulieren kann.

Zur Speicherung eines alphanumerischen Zeichens benötigt man bei den meisten FORTH-Systemen nur ein einziges Byte. Deshalb besprechen wir in diesem Kapitel auch FORTH-Wörter, mit denen man einzelne Bytes manipulieren kann. Die Operationen, die wir dabei vorstellen, können sowohl auf Zahlen als auch auf Zeichen angewendet werden. In einem 8-Bit-Byte kann man nämlich auch vorzeichenlose Integers im Bereich von 0 bis 255 speichern.

7.1 Zeichen - Bytemanipulationen

Dieser Abschnitt legt dar, wie man in FORTH alphanumerische Daten eingeben, ausgeben und verarbeiten kann. Wir werden auch sehen, wie man Text speichert und manipuliert. Da das FORTH-System Zeichen byteweise ablegt, werden wir uns auch mit FORTH-Wörtern beschäftigen, die uns die Bearbeitung einzelner Bytes erlauben.

C! und **C@** - Das FORTH-Wort **C!** entfernt zwei einfach genaue Zahlen vom Stack; der erste Stackeintrag wird als Adresse behandelt, während der zweite Stackeintrag eine vorzeichenlose Integer sein muß. Die acht niedrigstwertigen Bit dieser Integer werden dann an der angegebenen Speicheradresse abgelegt. Wir haben folgende Stack-Relation:

```
n a -->                                    (7-1)
```

C! funktioniert also fast genauso wie **!**, außer daß nur 8 Bits, (also ein Byte) gespeichert werden. Die Stack-Operationen sind jedoch bei beiden Wörtern gleich (eine einfach genaue Integer belegt auf dem Stack zwei Byte.) Mit dem Stack-Symbol "a" bezeichnen wir bekanntermaßen Adressen. Eine Adresse ist aber nichts anderes als eine vorzeichenlose einfach genaue Integer.

Zum Wiederauffinden gespeicherter Zeichen dient das FORTH-Kommando **C .** Es entfernt die oberste Zahl vom Stack und behandelt sie als Adresse. Das Byte, das an dieser Adresse gespeichert ist, wird geholt und als einfach genaue Integer auf den Stack gelegt. Da eine einfach genaue Integer doppelt so lang wie ein Byte ist, werden die acht höchstwertigen Bit der auf den Stack gelegten Zahl auf Null gesetzt. Die acht niedrigstwertigen Bit entsprechen jedoch der geholten Originalzahl. Die Stack-Relation lautet:

```
a --> n                                                  (7-2)
```

Die beiden Wörter **C!** und **C@** behandeln den Stack somit so, als ob einfach genaue Integers gespeichert bzw. geholt werden sollen. Der gespeicherte Wert belegt im Wörterbuch aber tatsächlich nur ein Byte.

CMOVE und **<CMOVE** - Bei der Arbeit mit Zeichen, die sich im Computerspeicher befinden, taucht oft die Notwendigkeit auf, ganze Speicherblöcke zu verschieben. Dazu müssen Sie sich nur einmal vorstellen, daß die im Arbeitsspeicher abgelegten Zeichen einen Absatz eines längeren Textes darstellen, den wir in einem selbstgeschriebenen Textprogramm verarbeiten wollen. In diesen Text wollen Sie nun - irgendwo in der Mitte - neues Textmaterial einfügen. Dazu wäre es bequem, mit einem einzigen Kommando die ganze Textzeile an eine andere Stelle des Arbeitsspeichers verschieben zu können, um Platz für das neue Textmaterial zu erhalten. Die zwei oben genannten FORTH-Wörter unterstützten uns bei dieser Aufgabe. Das erste - **CMOVE** - könnte zum Beispiel folgendermaßen eingesetzt werden:

```
34500 38000 100 CMOVE                                    (7-3)
```

Dieses Beispiel bewirkt, daß ein 100 Byte langer Block, der bei der Adresse 34500 beginnt, an eine neue Stelle im Arbeitsspeicher verschoben wird, und zwar beginnend bei Adresse 38000. Dadurch wird die Ausgangsinformation dupliziert, d.h., die Bytes, die zwischen Adresse 34500 und 34599 im Arbeitsspeicher stehen, wiederholen sich jetzt an den Adressen 38000 bis 38099. Bei dieser Art der Blockverschiebung bleiben die Ausgangsadressen also unverändert. Sie stehen jetzt jedoch zur Speicherung neuer Informationen zur Verfügung. Die Informationsverschiebung geschieht byteweise "von unten nach oben", d.h., zuerst wird das Byte an Adresse 34500 an die Adresse 38000 bewegt, dann verschiebt FORTH das Byte an Adresse 34501 nach 38001 usw. Die Reihenfolge, in der die einzelnen Bytes bewegt werden, scheint für den Benutzer uninteressant zu sein. Im vorhergehenden Beispiel ist sie es auch. Es gibt jedoch bestimmte Fälle, in denen der Benutzer die Details des Verschiebeprozesses beachten muß. Wenn wir z.B. folgendes schreiben:

```
34500  34550  100 MOVE                                  (7-4)
```

dann werden zwar wiederum 100 Byte verschoben, die Verschiebedistanz beträgt diesmal jedoch nur 50 Byte. Bei der ersten Bytebewegung überschreiben wir die Daten an Adresse 34550 unwiederbringlich. Die 50 Byte, die ab dieser Adresse beginnen, werden also de facto gar nicht mit verschoben. Eine Lösung für dieses Problem könnte darin bestehen, den 100-Byte-Block weiter als 100 Speicherstellen zu verschieben und dann die richtig kopierten Daten mit einem weiteren **CMOVE** an die gewünschte Stelle zu bringen. Glücklicherweise gibt es jedoch ein anderes FORTH-Wort, das dieses Problem löst. Dieses lautet **<CMOVE** und arbeitet im wesentlichen wie **CMOVE**, außer daß bei dem Verschieben mit den höheren Adressen begonnen wird. Deshalb bewirkt

```
34500 34550 100 <CMOVE                                  (7-5)
```

das Folgende: Erst wird der Inhalt der Adresse 34599 an der Speicheradresse 34649 wiederholt. Dann wird der Inhalt von Adresse 34598 verschoben, und zwar an Adresse 34648 usw. Somit werden die Daten an der Ausgangsposition der Verschiebeoperation nicht überschrieben, ehe sie nicht an die gewünschte Zielposition dupli-

ziert wurden. **CMOVE** und **<CMOVE** führen also dieselbe Funktion aus, gehen dabei jedoch unterschiedlich vor. Beide Wörter können natürlich auch dazu benutzt werden, Speicherblöcke von höheren in niedrigere Speicheradressen zu verschieben. Beachten Sie, daß **<CMOVE** noch kein Bestandteil von FORTH-79 ist. In MMSFORTH und anderen FORTH-Systemen ist es jedoch implementiert. Beide Wörter besitzen die Stack-Relation:

a_1 a_2 n --> (7-6)

Ein n Byte langer Block, der an Adresse a_1 beginnt, wird zu der neuen Startadresse a_2 verschoben.

MOVE - Auch dieses Wort dient zum Verschieben von Speicherblöcken im Computerspeicher und ähnelt **CMOVE**, außer daß beim Verschiebevorgang jeweils zwei Byte auf einmal kopiert werden. Wenn wir eingeben:

a_1 a_2 n MOVE

dann werden 2n Byte verschoben, beginnend bei Adresse a_1 mit dem Ziel a_2. **MOVE** ist Bestandteil von FORTH-79.

KEY - Mit dem FORTH-Kommando **KEY** können Sie alphanumerische Daten von der Tastatur eingeben. Bei Ausführung von **KEY** wird der ASCII-Code (vgl. Abschnitt 3-2) des nächsten von der Tastatur eingegebenen Zeichens als vorzeichenlose einfach genaue Integer auf den Stack gelegt. Beachten Sie, daß ASCII-Codes stets kleiner oder gleich 255 sind, so daß man sie auch als vorzeichenlose Integers in einem einzigen Byte darstellen kann. Das bedeutet, daß man die ASCII-Darstellung eines Zeichens mit den Speicherkommandos **C!** oder **C@** manipulieren kann. Wenn wir also eingeben:

KEY Z (RETURN) (7-7)

dann erhalten wir als obersten Stack-Eintrag die Zahl 90 als vorzeichenlose einfach genaue Integer, weil dies der ASCII-Code für den Buchstaben Z ist. Die Stack-Relation für KEY lautet:

```
--> u                                                    (7-8)
```

Wie Sie wissen, dient in Stack-Relationen der Buchstabe "u" zur Darstellung von vorzeichenlosen einfach genauen Integers. Bei Ausführung von **KEY** wird allerdings das nächste Zeichen von der Tastatur nicht auf dem Bildschirm ausgegeben (geechot). Das bedeutet, daß bei Ausführung von (7-7) der Benutzer auf seinem Bildschirm nichts zu sehen bekommt.

EMIT - Das FORTH-Kommando **EMIT** dient zur Ausgabe von Zeichen aufgrund ihres ASCII-Codes. Dazu entfernt **EMIT** eine vorzeichenlose einfach genaue Integer vom Stack und gibt das Zeichen aus, das gemäß der ASCII-Code-Tabelle dieser Zahl entspricht. Beachten Sie, daß der Stackwert zwischen 0 und 127 liegen muß. Durch Ausführung von

```
90 EMIT (RETURN)                                         (7-9)
```

erhalten wir so ein großes Z auf dem Bildschirm. **EMIT** hat folgende Stack-Relation:

```
u -->                                                    (7-10)
```

Wir wollen unsere neu gelernten Wörter gleich in einem Beispiel anwenden. In Abbildung 7-1 sehen Sie ein kleines Programm, mit dem Textmaterial in einen Array eingegeben werden kann. In diesen Array können Sie etwa einen kurzen Brief eingeben und später wieder ausdrucken oder den Text sogar mit einem - selbstgeschriebenen - Textprogramm bearbeiten. Zur Einrichtung von Byte-Arrays müssen wir die Verfahren aus dem letzten Kapitel anwenden.

Jetzt zu den Details des Programms in Abbildung 7-1. In Zeile 2 besorgen wir uns Speicherplatz für 1024 Byte, indem wir zuerst eine Variable **TEXT** vereinbaren und deren Speicherumfang dann mittels **ALLOT** ausdehnen. Wir können in diesem Array jetzt 1024 Zeichen speichern. Die ersten beiden Bytes, die **TEXT** ursprünglich durch **VARIABLE** zugewiesen wurden, benutzen wir dazu, die Anzahl der tatsächlich eingegebenen Zeichen in dem Array zu speichern.

```
 0 ( Einfache Ein- und Ausgabe von Zeichen )
 1 VARIABLE TEXT    1024 ALLOT
 2 (Zeicheneingabe)
 3 : SPEICHERE  CR  1024  1  DO   KEY   DUP   EMIT   DUP
 4              TEXT  I  2+  +  C!
 5              35  -  0=
 6              IF  LEAVE   I  1-  TEXT  !  THEN
 7        LOOP    CR    ;
 8
 9 (Gespeicherte Daten ausgeben)
10 : AUSGABE  CR    TEXT @   1  DO
11            TEXT  I  1+  +   C @
12            EMIT
13        LOOP    CR    ;
14
15
```

ABBILDUNG 7-1: Einlesen von Zeichendaten in Arrays

Das eigentliche "Speicherwort" beginnt in Zeile 3 und trägt den Namen **SPEICHERE.** Im wesentlichen besteht dieses Wort aus einer Schleife, die 1024mal wiederholt wird. Nach Eintritt in die Schleife wird das Wort **KEY** gerufen, wodurch das Programm anhält und wartet, bis der Benutzer eine Taste gedrückt hat. **KEY** wandelt dann diesen Tastendruck in den entsprechenden ASCII-Code um, den es auf den Stack legt. Wir duplizieren die Zahl und geben mittels **EMIT** das entsprechende Zeichen wieder aus, so daß der Benutzer auch sieht, was er eingegeben hat. Wieder duplizieren wir die Benutzereingabe. Dies ist der letzte Befehl von Zeile 3. In Zeile 4 sorgen wir dafür, daß das eingegebene Zeichen an der richtigen Stelle im Array **TEXT** abgelegt wird. Dazu besorgen wir uns die Startadresse von **TEXT** sowie den Laufindex der Schleife. Auf diesen Wert addieren wir 2 (da ja die ersten beiden Bytes von **TEXT** zur Speicherung der Zeichenzahl dienen), und nach Addition mittels **+** erhalten wir die richtige Adresse. An dieser speichern wir sodann mit **C!** die Benutzereingabe. Dieser Prozeß wiederholt sich bei jedem Schleifendurchgang, bis **TEXT** die gewünschten Daten enthält. Da es aber sein kann, daß der Benutzer nicht 1024 Zeichen eingeben, sondern vorzeitig aufhören will, überprüfen wir noch auf ein spezielles Endezeichen in Zeile 5. Dazu vereinbaren wir, daß die Eingabe in den Array **TEXT** beendet wird, sobald der Benutzer "#" eingibt; dieses Zeichen hat den ASCII-Code 35. Natürlich müssen Sie nicht unbedingt das # als Endezeichen ver-

wenden; jedes andere Zeichen ist möglich, allerdings ist zu bedenken, daß dieses Zeichen in normalem Textmaterial vermutlich nicht vorkommt. Nun zu Zeile 5, in der überprüft wird, ob der Benutzer die #-Taste gedrückt hat. Dazu subtrahieren wir 35 vom obersten Stack-Eintrag und überprüfen mit **0=** auf Gleichheit mit 0. Wenn **0=** ein "wahres" Flag hinterläßt, dann ist es Zeit, die Schleife zu beenden, wofür **LEAVE** sorgt. Nach Verlassen der Schleife müssen wir jedoch noch die Anzahl der eingegebenen Zeichen an den Anfang des Arrays schreiben. Dazu erniedrigen wir den Schleifenindex um 1 (**I 1-**). Diese Zahl wird ganz am Anfang des Speicherbereichs für **TEXT** abgelegt. Jetzt sind wir mit **SPEICHERE** fertig, und der Array **TEXT** enthält die Zeichen, die der Benutzer von der Tastatur eingegeben hat. In den ersten beiden Bytes des Arrays steht außerdem noch, wie viele Zeichen er eingegeben hat.

Die Zeilen 9 bis 13 der Abbildung 7-1 zeigen das Wort **AUSGABE**, mit welchem man den Text im Array **TEXT** auf dem Bildschirm anzeigen kann. Das Wort besorgt sich zuerst in Zeile 10 die Anzahl der auszugebenden Zeichen und legt sie auf den Stack. Dieser Wert dient als Testwert für eine DO-Schleife, deren Schleifenkörper in Zeile 11 beginnt. Hier wird der ASCII-Code jedes im Array gespeicherten Zeichens geholt; **EMIT** wandelt diesen Code dann in das entsprechende Zeichen um und gibt es auf dem Bildschirm aus. Dieses Verfahren wiederholt sich bei jedem Eintrag im Array **TEXT**. Wir wollen einmal sehen, ob es uns gelingt, Textmaterial an beliebiger Stelle in den Array **TEXT** einzufügen. Dies ist eine Fähigkeit, über die jedes Textverarbeitungsprogramm verfügen muß. Das Programm aus der Abbildung 7-2 löst diese Aufgabe.

Wenden wir uns nun diesem neuen Programm zu, dem wir den Namen **EINFGN** gegeben haben. Wir gehen davon aus, daß der Block aus der Abbildung 7-1 geladen ist und bereits Daten in den Array **TEXT** eingegeben wurden. Im Programm der Abbildung 7-2 nehmen wir fernerhin an, daß der einzufügende String höchstens 10 Zeichen lang ist. Die Einfügezeichen werden in einem separaten Array zwischengespeichert, der den Namen **EINFBUF** trägt und in Zeile 1 der Abbildung 7-2 vereinbart wird. Wie schon in den anderen Programmen dienen die ersten beiden Bytes von **EINFBUF** dazu, die Anzahl der tatsächlich einzufügenden Zeichen zu speichern. Die Zeichen selbst stehen dann im Rest des Arrays. Wir können die Zahl der Zeichen, die eingefügt werden können, beliebig vergrößern, indem wir diesen Zwischenspeicher erweitern.

```
 0 ( Einfuegen eines Strings )
 1 VARIABLE EINFBUF   10 ALLOT  VARIABLE POINT
 2 : EINFGN  CR  ." AB WO SOLL EINGEFUEGT WERDEN? " #IN  POINT !
 3             CR ." BIS ZU 10 EINFUEGEZEICHEN EINGEBEN "  CR
 4             10  1   DO   KEY  DUP  EMIT  DUP
 5             EINFBUF  I 1+   +    C!
 6             EINFBUF   I    !    35   -   0=
 7             IF LEAVE    THEN   I 1-   EINFBUF !
 8      LOOP        CR
 9        TEXT  2  +   POINT  @   +   DUP   EINFBUF @  +
10           TEXT @  POINT   @    -   <MOVE
11        EINFBUF  2  +  TEXT   2  +   POINT  @    +
12           EINFBUF   @    CMOVE
13        EINFBUF  @  WORDS    +!        ;
14
15
```

ABBILDUNG 7-2: Einfügen von Textmaterial in den Array **WORDS**

In der Fachsprache der Programmierer werden solche Zwischenspeicher auch Puffer genannt. Wir benötigen noch eine zweite Variable, die wir **POINT** nennen und in der wir uns die Position merken, an der der Text in den Array **TEXT** eingefügt werden soll; diese Variable wird ebenfalls in Zeile 1 vereinbart.

Das eigentliche Einfügeprogramm mit dem Namen **EINFGN** fragt als erstes den Benutzer nach der Stelle, an der die neuen Zeichen in den bereits existierenden Text eingefügt werden sollen. Er soll dazu eine einfach genaue Integer eingeben, die wir in der Variablen **POINT** aufheben. Als nächstes wird - in Zeile 3 - der Benutzer aufgefordert, den einzufügenden Text (maximal 10 Zeichen) einzugeben; wie auch bereits bei dem letzten Programm, wird dieser Text wieder durch das "#" abgeschlossen. Die Zeilen 4 bis 8 stellen eine Programmschleife dar, die den Einfügetext einliest und im Puffer **EINFBUF** zwischenspeichert. Wir folgen dabei den Verfahrensschritten, die wir bereits aus Abbildung 7-1 kennen. Nach Verlassen der Schleife steht der einzufügende Text fest, und der eigentliche Einfügevorgang kann beginnen. Dazu beschaffen wir uns zuerst im Array **TEXT** genügend Platz, indem wir Teile dieses Textes im Arbeitsspeicher verschieben. Die erste zu verschiebende Adresse ergibt sich aus der Anfangsadresse von **TEXT**, auf die wir 2 und den in der Variablen **POINT** gespeicherten Wert addieren. Die Addition der Zwei ist nötig, weil die ersten beiden Bytes des

Arrays für die Zeichenzahl im Array reserviert sind. Als nächstes berechnen wir die Endadresse des zu verschiebenden Blockes. Dazu duplizieren wir die soeben berechnete Anfangsadresse und addieren darauf die Anzahl Zeichen, die aus dem Puffer **EINFBUF** eingefügt werden sollen. Diese finden wir natürlich in den ersten beiden Bytes von **EINFBUF**. Bezogen auf das Stack-Diagramm 7-6 haben wir jetzt die Werte a_1 und a_2 auf den Stack gelegt. Es verbleibt nun noch, die Gesamtzahl der zu verschiebenden Zeichen zu berechnen. Wir erhalten diese Zahl, indem wir von der Anzahl der in **TEXT** gespeicherten Zeichen die Anzahl der Zeichen abziehen, die vor der Einfügeposition in **POINT** stehen. Zeile 10 der Abbildung 7-2 führt die nötigen Berechnungen aus und sorgt mittels **<CMOVE** dafür, daß die Verschiebung stattfindet.

Jetzt ist im Array **TEXT** eine Lücke entstanden, die groß genug ist, um den Einfügetext aufzunehmen. Beachten Sie, daß diese "Lücke" nicht leer ist, sondern nach wie vor die alten Daten enthält, die jedoch dupliziert wurden und so ohne Probleme überschrieben werden können. Um nicht versehentlich unsere Verschiebedaten zu zerstören, haben wir das Wort **<CMOVE** verwendet anstelle des einfachen **CMOVE**.

Wir können jetzt die Daten aus unserem Puffer in die Lücke im Array **TEXT** übertragen. Nach Ausführung der Zeilen 11 und 12, aber ehe das Wort **CMOVE** zur Ausführung gelangt, befinden sich folgende Informationen als einfach genaue Integers auf dem Stack: Die Adresse des Puffers plus 2 als dritter Stack-Eintrag; dabei handelt es sich um die Anfangsadresse des Datenblockes für die nachfolgende Verschiebeoperation. Die Zieladresse ist der zweite Stackeintrag; man erhält sie aus der Anfangsadresse von **TEXT**, erhöht um 2, sowie der Zahl, die in **POINT** gespeichert ist. An oberster Stack-Position finden wir schließlich die Anzahl der Zeichen, die im Puffer **EINFBUF** gespeichert sind und die eingefügt werden sollen. Damit ist alles für die Einfügeoperation vorbereitet, und wir können - entweder mit **CMOVE** oder mit **<CMOVE** - den Verschiebevorgang auslösen. Welche der beiden Blockoperationen wir wählen, ist in diesem Falle bedeutungslos, da sich die Verschiebedaten nicht irrtümlich selbst überschreiben können.

Nachdem der Verschiebevorgang abgeschlossen ist, müssen wir noch die Zeichenzahl aktualisieren, die wir uns am Anfang von **TEXT** merken. Dies besorgt Zeile 13 des Programms. Die Ausgangszahl - also die Anzahl Zeichen vor dem Einfügevorgang - wird um die Anzahl der Einfügezeichen erhöht; letztere finden wir am Anfang des

Puffers **EINFBUF.** Wir haben in Zeile 13 das Wort +! angewendet und so die beiden Vorgänge Addieren und Speichern in möglichst knapper Form bewirkt.

An dieser Stelle ist einige Vorsicht geboten; Sie müssen aufpassen, daß Sie nicht versehentlich zu viele Zeichen einfügen. Der Array **TEXT** kann ja nur insgesamt 1024 Zeichen aufnehmen. Wenn Sie nun versuchen, in **TEXT** neue Zeichen einzufügen, so daß sich eine Gesamtzeichenzahl von mehr als 1024 ergibt, dann kann es sein, daß Sie damit Teile des FORTH-Wörterbuches überschreiben. Eventuell müssen Sie dann Ihr FORTH-System neu starten und verlieren dabei alle Ihre Daten. Sie können zwar mit dem Programm von Abbildung 7-1 nicht mehr als 1024 Zeichen in einen Array schreiben. Nichts und niemand hindert Sie jedoch daran, durch wiederholte Anwendung von **EINFGN** eine größere Anzahl an Zeichen einzufügen und somit die Grenze von 1024 zu überschreiten. Um dies zu verhindern, sollte man das FORTH-Wort **EINFGN** um eine zusätzliche Überprüfung erweitern, in der nachgesehen wird, ob die Summe der einzufügenden Zeichen und der bereits vorhandenen Zeichen den Wert 1024 übersteigt.

TYPE - Mit dem FORTH-Wort **TYPE** können Zeichenketten (Strings) ausgegeben werden, die im Speicher des Computers stehen. Ein Beispiel für die Anwendung von **TYPE** sieht folgendermaßen aus:

```
TEXT 2 + 15 TYPE
```

Dies führt dazu, daß, beginnend bei der Adresse **TEXT** plus 2, 15 Zeichen aus dem Array ausgegeben werden. **TYPE** besitzt folgende Stack-Relation:

```
a n -->
```

7.2 Weitere Wörter für die Zeicheneingabe

Wir machen Sie jetzt mit einigen weiteren FORTH-Kommandos vertraut, die zur Eingabe von Zeichendaten dienen. Sie können sie in Ihre eigenen Definitionen mit einschließen. Das FORTH-System selbst macht heftigen Gebrauch von diesen Wörtern, da mit ihrer Hilfe die Eingabe von Zeichen in das System organisiert wird.

EXPECT - Mit Hilfe des FORTH-Wortes **EXPECT** kann man eine ganze Folge von Zeichen (einen String) eingeben. Eine typische Anwendung sieht etwa so aus:

```
    TEXT 10 EXPECT                                        (7-11)
```

Dabei gehen wir davon aus, daß der Array **TEXT** vor der Ausführung von (7-11) vom Benutzer vereinbart worden ist. Wenn wir (7-11) eingeben, dann unterbricht das System seine Arbeit und wartet darauf, daß der Benutzer Zeichen eingibt. In diesem Fall werden 10 Zeichen erwartet (englisch: "to expect"), da dies die Zahl war, die das Wort **EXPECT** bei seinem Aufruf auf dem Stack vorfand. Die Zeichen werden in den Array **TEXT** eingegeben, wobei mit der Startadresse von **TEXT** begonnen wird, da sich diese nach Aufruf des Arraynamens auf dem Stack befindet. Das System fährt so lange mit der Zeicheneingabe fort, bis der Benutzer entweder 10 Zeichen eingegeben oder bis er die Return-Taste gedrückt hat. Das Wort **EXPECT** hat folgende Stack-Relation:

```
    a n -->                                               (7-12)
```

In Abbildung 7-3 sehen Sie ein einfaches Beispielprogramm, das **EXPECT** einsetzt. In Zeile 1 vereinbaren wir einen Array, der zur Aufnahme von Zeichen bestimmt ist und den Namen **LETTERS** trägt. Darauf folgt, beginnend mit Zeile 2, die Definition des Worts **CHAR**; in diesem Wort werden zuerst mittels **EXPECT** 10 Zeichen (oder weniger, je nach Benutzereingabe) im Array **LETTERS** abgespeichert. Anschließend wird die Information wieder ausgegeben, und zwar sowohl die eingegeben Zeichen als auch deren ASCII-Code. Dazu benutzen wir sowohl das Wort **EMIT** als auch das Punktkommando. Um sowohl das Zeichen als auch dessen ASCII-Code ausgeben zu können, müssen wir zuvor jedes Zeichen mittels **DUP** duplizieren.

Gemäß seiner Vereinbarung können wir in dem Array **LETTERS** genau 10 Zeichen speichern. Oft weisen wir zwei zusätzliche Bytes an Speicherplatz in einem Array zu, um darin als einfach genaue Integer die Anzahl der Zeichen in einem Array zu speichern. Andere FORTH-Systeme verfahren wiederum anders, in ihnen werden ein oder zwei Sonderzeichen hinter die eingegebene Textinforma tion geschrieben, an denen das System das Ende eines Strings erkennt. Diese Endemarkierung wird automatisch von **EXPECT** angefügt.

```
0 ( Einfaches Beispiel für EXPECT )
1 VARIABLE LETTERS    8  ALLOT
2 : CHAR    LETTERS   10 EXPECT   CR
3     10   0  DO   LETTERS  I  +   C  DUP   EMIT
4      LOOP     ;
5
```

ABBILDUNG 7-3: Anwendungsbeispiel für **EXPECT**

Vergewissern Sie sich anhand Ihrer Sprachbeschreibung, welchen Konventionen Ihr **EXPECT** folgt, und vergessen Sie dann nicht, entsprechend ausreichenden Speicherplatz für Strings bereitzustellen.

WORDS, >IN, HERE und **BLK** - Wenn Sie von der Tastatur Ihre Befehle an das FORTH-System geben, dann erscheinen Ihre Eingaben dem System erst einmal als eine unzusammenhängende Folge von Zeichen. Um zu "verstehen", was Sie meinen, muß das System diesen Zeichenstrom als erstes in sinnvolle Einheiten aufteilen, d.h., es muß erkennen, welche Ihrer Eingaben FORTH-Wörter sind und bei welchen es sich um Daten handelt, die auf den Stack gelegt werden sollen. Dazu sucht das System nach speziellen Begrenzungszeichen oder Trennzeichen, an denen es die Wortgrenzen erkennen kann. Das wichtigste Trennzeichen in FORTH ist natürlich das Leerzeichen, es gibt aber auch noch andere Trenner. Mit dem FORTH-Kommando **WORD** können Sie außerdem vereinbaren, welches Zeichen als Trennzeichen fungieren soll. **WORD** macht aber noch mehr: Wenn es gerufen wird, liest es so lange Zeichen von der Tastatur, bis das vereinbarte Trennzeichen auftaucht. Betrachten Sie dazu folgendes Beispiel:

```
32 WORD                                        (7-13)
```

WORD entfernt die 32 vom Stack und merkt sich, daß es sich dabei nun um ein Trennzeichen handelt. Bei der 32 handelt es sich jedoch um den ASCII-Code für ein Leerzeichen. Hätten wir anstelle von 32 den Wert 35 auf den Stack gelegt, dann wäre das "#" neues Trennzeichen in unserem System.

Nachdem es sich nun das (eventuell neue) Trennzeichen gemerkt hat, liest **WORD** so lange Zeichen von der Tastatur, bis es auf dieses Trennzeichen stößt. Die eingelesenen Zeichen werden gespeichert, und die Anfangsadresse des Speicherbereichs, in dem sie abgelegt wurden, wird von **WORD** auf den Stack gelegt. Zusätzlich zu den eingegebenen Zeichen enthält das erste Byte an dieser Adresse eine Integer (die allerdings nur ein Byte lang ist), welche die Anzahl der eingegebenen Zeichen angibt. Denken Sie daran, daß **WORD** nur so lange Ihre Eingabedaten von der Tastatur speichert, bis es auf das Trennzeichen stößt. Wenn Sie Ihre Eingaben jedoch mit einer Folge von Trennzeichen einleiten, dann werden diese ignoriert. **WORD** beginnt also erst dann mit der Speicherung, wenn Sie ein Nicht-Trennzeichen von der Tastatur eingeben, und hört damit auf, wenn danach wieder ein Trennzeichen erscheint.

Zum "Wortschatz" von FORTH-79 gehört auch das Wort **HERE**, welches die Adresse des nächsten freien Wörterbucheintrags liefert. Bei Ausführung von **WORD** wird der eingelesene String ab der Adresse gespeichert, die man über **HERE** erhält. (Hier können sich einzelne FORTH-Systeme voneinander unterscheiden; Sie sollten die Einzelheiten in Ihrem Systemhandbuch überprüfen.)

Tatsächlich ist die Arbeitsweise von **WORD** komplizierter, als wir es bisher dargestellt haben. Nehmen wir an, der Benutzer gibt eine Folge von Wörtern ein, wobei das Leerzeichen als Trenner fungiert. So, wie wir **WORD** bisher geschildert haben, würde es nur das erste Wort speichern, da der Einleseprozeß beim ersten Trennzeichen abgebrochen wird. Es gibt jedoch ein weiteres FORTH-Wort, mit dem diese Situation geändert werden kann. Es lautet **>IN** und ist der Name einer Variablen in FORTH-79. Die unter **>IN** gespeicherte Zahl gibt an, wieviel Zeichen der Benutzereingabe FORTH überlesen soll, ehe es durch **WORD** die Eingabekette des Benutzers verarbeitet. Falls wir in **>IN** beispielsweise den Wert 3 stehen haben, dann werden die ersten drei Zeichen, die der Benutzer auf der Tastatur tippt, ignoriert, wenn wir **WORD** aufrufen. Die in **>IN** gespeicherte Zahl kann sich zwischen 0 und 1023 bewegen.

Wir müssen noch ein weiteres wichtiges FORTH-Wort besprechen. Bisher sind wir davon ausgegangen, daß die Eingaben an das System von der Tastatur kommen. Dies muß aber nicht so sein; genausogut können die Eingaben von der Diskette stammen. Wenn also **WORD** gerufen wird, dann müssen Sie zuerst dem FORTH-System mitteilen, von welcher Quelle die Eingabedaten kommen. Diese Information wird in einer anderen Standardvariablen von FORTH gespeichert,

die den Namen **BLK** trägt. Falls die in **BLK** gespeicherte Zahl gleich 0 ist, dann geht FORTH davon aus, daß die Eingaben an das System von der Tastatur kommen. (Die Details von Diskettenoperationen werden wir erst im nächsten Kapitel kennenlernen.)

```
0 (Einfaches Beispiel mit WORD)
1 : SCAN    0   BLK !   5 >IN  ! 32  WORD  C@   CR
2     1 +          1  DO  HERE  I  + C@      EMIT LOOP
3     CR CR        QUIT   ;
4
5
```

ABBILDUNG 7-4: Ein Beispiel für die FORTH-Wörter **WORD,>IN,HERE** und **BLK**

Am Beispiel von Abbildung 7-4 wird die Arbeitsweise der soeben vorgestellten FORTH-Wörter deutlicher. Das in dem Beispiel definierte Wort mit dem Namen **SCAN** legt als erstes fest, daß die Eingaben von der Tastatur kommen. Dies geschieht, indem wir in der Variablen **BLK** eine 0 speichern. Als nächstes vereinbaren wir, daß **WORD** die nächsten 5 Zeichen der Benutzereingabe übergehen soll. Der Grund dafür wird gleich klar werden. Wir bewirken dies, indem wir in der Variablen **>IN** eine 5 speichern. Dann rufen wir **WORD**, wobei wir durch eine auf den Stack gelegte 32 dafür sorgen, daß das Leerzeichen als Trennzeichen gilt. Nachdem **WORD** seine Aufgabe erledigt und die Eingabezeichen untersucht hat, findet sich die Adresse aus **HERE** auf dem Stack. Die Zahl, die an dieser Stelle gespeichert wird, besorgen wir uns nun mittels **C** . Sie entspricht der Anzahl der eingelesenen Zeichen. Auf diese Anzahl addieren wir 1 und haben somit alle Vorbedingungen für den Eintritt in eine Schleife erfüllt, die uns die eingelesenen Zeichen ausgeben soll. Beachten Sie, daß wir als Anfangsadresse dabei die Variable **HERE** angegeben haben. Wir beenden das Programm mit dem Wort **QUIT.** Würden wir diese Vorsichtsmaßnahme unterlassen, dann würden alle sonst noch bei Ausführung von **WORD** eingegebenen Zeichen anschließend als Eingaben in das FORTH-System interpretiert werden.

Angenommen, wir laden den Block aus Abbildung 7-1 und tippen anschließend ein:

SCAN WEISST DU WIEVIEL STERNLEIN STEHEN (RETURN)   (7-14a)

Daraufhin bekommen wir folgendes auf dem Bildschirm zu sehen:

```
WEISST                                                    (7-14b)
```

Wie Sie wissen, ist in der Variablen **>IN** eine 5 gespeichert; deshalb werden die ersten 5 Zeichen des Eingabestroms von **SCAN** übergangen. Dies bewirkt, daß das Wort **SCAN** selbst überlesen und erst das erste Wort des Satzes ausgegeben wird. Wenn wir (7-14a) eingeben, dann sorgt der Befehl **SCAN** dafür, daß das Programm aus Abbildung 7-4 aufgerufen wird. Zu diesem Zeitpunkt befindet sich aber noch der ganze eingegebene Text einschließlich des Schlüsselwortes **SCAN** im Eingabestrom. Deshalb vereinbaren wir in Zeile 1 unseres Programms, daß die ersten fünf Zeichen übergangen werden sollen. Würden wir dies unterlassen, dann würde das Programm aus Abbildung 7-4 stets nur das Wort **SCAN** ausgeben und sich um weitere Benutzereingaben nicht kümmern. Hätten wir schließlich auf das **QUIT** in Zeile 3 verzichtet und gäben 7-14a ein, dann würde das FORTH-System versuchen, das Wort **DU** als FORTH-Kommando zu interpretieren und dabei vermutlich auf einen Fehler laufen.

QUERY - Im vorigen Beispiel mußten wir den Wert von **>IN** auf 5 setzen, damit die vier Buchstaben des selbstdefinierten Wortes **SCAN** nicht versehentlich beim Aufruf dieses Wortes mit eingelesen werden. Es gibt jedoch ein spezielles FORTH-Wort, mit dem dieses vermieden werden kann; es lautet **QUERY**. Wenn wir dieses Wort rufen, dann werden die nächsten 80 von der Tastatur eingegebenen Zeichen im Eingabepuffer gespeichert. Im Puffer befinden sich somit nur die Zeichen, die nach Aufruf des Wortes **QUERY** eingegeben wurden, nicht aber die Zeichenfolge "QUERY" selbst. **QUERY** stellt den Einlesevorgang ein, wenn entweder 80 Zeichen vom Benutzer getippt oder die Return-Taste gedrückt wurde. Falls sowohl in **BLK** als auch in **>IN** eine 0 gespeichert ist, dann liest **WORD** seine Informationen aus dem Eingabepuffer. Wir brauchen also nur das Programm in 7-4 so abzuändern, daß wir die 5 in Zeile 2 durch eine 0 ersetzen und vor der 32 in dieser Zeile das Wort **QUERY** einfügen. Jetzt befindet sich nach Aufruf von **SCAN** das Wort "SCAN" selbst nicht im Eingabepuffer, wenn wir **WORD** rufen. Das Programm gibt also auch nach dieser Veränderung das korrekte Ergebnis aus.

7.3 Stringverarbeitung

In diesem Abschnitt lernen wir einige FORTH-Wörter kennen, die bei der Bearbeitung von Strings sehr nützlich sind. Wiederum befinden sich nicht alle diese Wörter im FORTH-79-Standard, sie sind jedoch Teil von MMSFORTH und anderen FORTH-Systemen. Schlagen Sie in Ihrem Handbuch nach, um herauszufinden, welche Befehle Ihr FORTH-System kennt und gegebenenfalls, welchen Namen die betreffenden Wörter dort haben.

7.3.1 Stringkonstanten, Variable und Arrays

Ebenso wie bei anderen Datentypen kann man auch Konstanten, Variable und Arrays definieren, die zur Speicherung von Strings dienen. Bereits in Abschnitt 7-1 haben wir besprochen, wie man dies mit Wörtern aus dem "Wortschatz" von Standard-FORTH erreichen kann. Wir wenden uns nun einigen nichtstandardisierten Wörtern zu, die bequemer in der Handhabung sind. Zur Vereinbarung einer String-Konstante dient das Wort **$CONSTANT.** In vielen Programmiersprachen, einschließlich BASIC, dient das Dollar-Zeichen zur Markierung von Stringausdrücken. Hier ein Beispiel für die Anwendung von **$CONSTANT:**

```
$CONSTANT EINE KONSTANTE"                              (7-15)
```

Beachten Sie, daß in dieser Befehlsfolge nur ein einzelnes Anführungszeichen vorkommt. In diesem Fall haben wir eine Stringkonstante vereinbart, der wir den Namen **EINE** gegeben haben. Diese Konstante enthält den String **KONSTANTE.** Im ersten Byte von **EINE** findet man eine ein Byte lange Integer, die die Länge des abgespeicherten Strings angibt. Im Beispiel (7-15) ist die Länge des Datenbereichs für den Wörterbucheintrag **EINE** genau 9 Byte. Bei Ausführung von **EINE** wird die Adresse des ersten Bytes auf den Stack gelegt. Dieses Verfahren unterscheidet sich also von dem, das bei numerischen Konstanten üblich ist. Zur Vereinbarung einer Stringvariablen dient entsprechend das Wort **$VARIABLE.** Ein Beispiel für seinen Einsatz:

20 $VARIABLE ABC (7-16)

In diesem Fall vereinbaren wir einen Wörterbucheintrag mit dem Namen **ABC**, der für 21 Byte Platz hat. Bei Einrichtung der Variablen werden alle diese Bytes mit dem Wert 0 besetzt. Auch hier dient das erste Byte oftmals dazu, die Länge des Strings festzuhalten, der in der Variablen **ABC** gespeichert ist. Dies geschieht jedoch nicht automatisch. Der Programmierer muß selbst dafür sorgen, daß die notwendigen Informationen festgehalten werden. Die Stack-Relation für **$VARIABLE** lautet:

n --> (7-17)

Stringvariable und Stringkonstanten werden in gleicher Weise verwendet. Im Unterschied zu einer Variablen wird bei einer Stringkonstante jedoch der Konstantenwert zu dem Zeitpunkt bestimmt, in dem die Konstante vereinbart wird. Es wird jedoch später noch deutlich werden, daß wir - ebenso wie bei Stringvariablen - auch bei Stringkonstanten den String mit speziellen FORTH-Wörtern lesen und verändern können. Auch dies unterscheidet sich von der Behandlung, die numerische Konstanten und Variable in FORTH erfahren.

Schließlich gibt es noch ein Wort, mit dem man in FORTH-Stringarrays vereinbaren kann; es lautet **$ARRAY**. Aus Kapitel 7-1 wissen wir bereits, daß man Stringarrays auch mit **ALLOT** vereinbaren kann. Die Handhabung von **$ARRAY** ist jedoch bequemer, allerdings ist dies kein Wort des FORTH-79-Standards. Hier ein Anwendungsbeispiel:

20 10 $ARRAY LETTERS (7-18)

In diesem Beispiel haben wir einen Array mit dem Namen **LETTERS** vereinbart, der aus 11 Elementen besteht. Jedes dieser 11 Elemente kann bis zu 21 Byte speichern. **$ARRAY** hat die Stack-Relation

n_1 n_2 --> (7-19)

Beachten Sie, daß n_2+1 gleich der Anzahl der Elemente im Array ist, während n_1+1 gleich der Anzahl Bytes ist, die in jedem Arrayelement gespeichert werden können (die Stringlänge). Alle Arrayelemente werden bei Definition des Arrays mit 0 vorbesetzt. Wenn wir die Adresse des ersten Bytes vom dritten String des Arrays **LETTERS** auf dem Stack haben wollen, dann geben wir ein:

```
3 LETTERS
```

Auch zweidimensionale Stringarrays sind möglich; man vereinbart sie mit **2$ARRAY.** Diese Vereinbarung muß folgende allgemeine Form haben:

n_1 n_2 n_3 2$ARRAY TABELLE (7-20)

Hier wird ein zweidimensionaler Array vereinbart, der n_2+1 Zeilen und n_3+1 Spalten hat. Jedes Element in dieser "Tabelle" ist seinerseits ein String, der maximal n_1 Byte lang sein kann. n_1 darf in diesem Fall den Wert 254 nicht übersteigen. Wir haben folgende Stack-Relation:

n_1 n_2 n_3 --> (7-21)

Auch hier werden alle Arrayelemente mit dem Wert 0 vorbesetzt. Um die Adresse des Strings in Zeile 3 und Spalte 5 von **TABELLE** auf dem Stack zu erhalten, müssen wir eingeben:

```
3 5 TABELLE
```

Denken Sie daran, daß bei der Numerierung von Zeilen und Spalten in einem Array mit Null begonnen wird.

7.3.2 Stringverarbeitung

Wir stellen als nächstes dar, wie man Strings speichern, auslesen und manipulieren kann. Als erstes lernen wir das Wort **$!** kennen. Angenommen, wir haben bereits Daten in einer Stringvariablen oder Stringkonstanten mit dem Namen **ABC** und wollen jetzt, daß diese Daten in eine andere Stringvariable dupliziert werden, welche den Namen **TOP** trägt. Dies erreichen wir mit folgender Wortfolge:

```
ABC TOP $!                                              (7-22)
```

Die Verfahren bei der Arbeit mit Strings unterscheiden sich von denen, die wir von Zahlen her gewöhnt sind. Das kommt daher, weil es meistens sinnlos ist, einen ganzen String auf den Stack zu legen. Falls der in der Variablen **ABC** länger als der ist, der sich in **TOP** befindet, dann werden Teile des Wörterbuches überschrieben. Dies müssen Sie unbedingt vermeiden. Zur Ausgabe eines Strings benutzen wir den Befehl **$.**. Wir können den in **TOP** gespeicherten String mit folgendem Befehl ausgeben:

```
TOP $.                                                  (7-23)
```

$. erwartet also die Adresse eines Strings auf dem Stack, weswegen seine Stack-Relation lautet:

```
a -->                                                   (7-24)
```

Erinnern Sie sich daran, daß wir mit "a" in Stack-Relationen Adressen bezeichnen. Im Arbeitsspeicher des FORTH-Systems gibt es einen speziellen Speicherbereich, das Scratch pad (wörtlich etwa "Schmierzettel"). Hierbei handelt es sich um eine Gruppe von Speicherstellen, die zur temporären Zwischenspeicherung von Stringdaten und anderen Informationen dient. Die Anfangsadresse dieses Bereichs ist in der FORTH-79-Variablen **PAD** gespeichert. Bei wachsendem Wörterbuch ändert sich auch die Lage dieses Bereichs im Arbeitsspeicher. Die Ausführung des Wortes **PAD** führt

dazu, daß die Adresse des ersten Bytes in diesem speziellen Arbeitsbereich auf den Stack gelegt wird.

Manchmal wollen wir einen neuen String aus zwei bereits existierenden Strings formen, indem wir diese hintereinanderschreiben. Man nennt diese Operation Verkettung oder auch Konkatenation. Das FORTH-Wort, das eine Konkatenation zweier Strings bewirkt, lautet **$+** und funktioniert folgendermaßen:

ABC TOP $+ (7-25)

Dadurch wird der in **TOP** gespeicherte String an den in der Variablen **ABC** gespeicherten String angehängt. Das Ergebnis der Verkettung finden wir in **PAD**. Deswegen hinterläßt die Operation **$+** auch die Adresse von **PAD** als Ergebnis auf dem Stack. Bei der Verkettung von Strings sollten Sie mit Vorsicht vorgehen. Das Ergebnis der Verkettung wird länger als jeder der beiden Ausgangsstrings. Versuchen Sie deshalb nicht, ein Verkettungsergebnis in einem Wörterbucheintrag zu speichern, der dafür zu klein ist. Für das Wort **$+** erhalten wir die Stack-Relation:

$$a_{\$1} \ a_{\$2} \dashrightarrow a_{pad} \qquad (7\text{-}26)$$

Man kann einen String auch direkt von der Tastatur eingeben, indem man sich des Wortes **$"** bedient. Hier ein Beispiel:

$" WEISST DU WIEVIEL STERNLEIN STEHEN" (7-27)

Beachten Sie das Leerzeichen nach dem ersten Anführungszeichen. Der einzugebende String wird von einem zweiten Anführungszeichen abgeschlossen. Wenn wir das Beispiel (7-27) tippen und anschließend RETURN drücken, dann wird der eingegebene String im Arbeitsspeicher abgelegt, wobei als Ergebnis des **$"** die Anfangsadresse des Strings auf den Stack gelegt wird. Sollte **$"** in der Definition eines eigenen FORTH-Wortes vorkommen, dann wird der String im nächsten verfügbaren Wörterbucheintrag abgelegt.

Gelegentlich ist es wünschenswert, daß ein Programm den Benutzer zur Eingabe eines Strings auffordert. Dies erreichen wir mit dem Wort **IN$**. Bei Ausführung dieses Wortes hält das System an und bringt ein Fragezeichen auf den Bildschirm. Der Programmbenutzer gibt dann einen String ein, welchen er durch RETURN abschließt. Dieser String wird in **PAD** abgelegt, wobei die Adresse von **PAD** auf den Stack kommt. Denken Sie daran, daß es sich bei diesem Speicherbereich nur um einen temporären Speicher handelt; Sie sollten so bald wie möglich die dort zwischengespeicherten Daten an eine andere Stelle bringen, da durch nachfolgende Programmschritte der "Schmierzettel" überschrieben werden kann.

Gelegentlich wollen wir in einem Programm auf einzelne Zeichen in einem String zugreifen. Diese Notwendigkeit kann z.B. dann auftauchen, wenn wir einen String in eine andere Variable übertragen und dabei sichergehen wollen, daß wir nicht versehentlich das Wörterbuch überschreiben. In diesem Fall müssen wir die Anzahl der zu kopierenden Zeichen beschränken. Dafür können mehrere Wörter eingesetzt werden. Eines davon lautet **LEFT$** und wird so eingesetzt:

```
ABC 10 LEFT$
```

Diese Wortfolge führt dazu, daß die Adresse von **ABC** und die einfach genaue Integer 10 auf dem Stack stehen. **LEFT$** kopiert dann die ersten 10 Zeichen des Strings **ABC** in den temporären Arbeitsbereich und hinterläßt dessen Adresse, also den Wert von **PAD**. Wir haben folgende Stack-Relation:

$$a_{\$}\ n \dashrightarrow a_{pad} \qquad (7\text{-}28)$$

Es gibt noch zwei weitere Wörter, die mit **LEFT$** inhaltlich verwandt sind; eines davon heißt **RIGHT**. Dieses funktioniert ähnlich wie **LEFT$**, überträgt jedoch die letzten n Zeichen des Strings in den Arbeitsbereich. Falls es nötig ist, Teile aus der Mitte eines Strings zu kopieren, verwendet man das Wort **MID$**. Es hat folgende Stack-Relation:

$$a_{\$}\ n_1\ n_2 \dashrightarrow a_{pad} \qquad (7\text{-}29)$$

In diesem Fall werden n_2 Zeichen des Strings an der Adresse a_S in den Arbeitsbereich übertragen. Der Kopiervorgang beginnt dabei mit dem n_1-ten Zeichen, also n_1 Zeichen vom Anfang des Strings weg. Die Wörter **LEFT$**, **RIGHT$** und **MID$** ändern an dem ursprünglichen String nichts.

Gelegentlich ist es wünschenswert, die Zeichenketten auszutauschen, die in zwei Stringvariablen abgespeichert sind. Dies bewirkt man mit dem Wort **$XCG**; Sie sollten hierbei allerdings aufpassen, damit Sie nicht einen String an einen dafür zu kleinen Speicherplatz bewegen. Beim Austauschen von Strings wird ebenfalls der temporäre Zwischenspeicher benutzt. Die Stack-Relation für **$XCG** lautet:

$$a_{\$1}\ a_{\$2} \dashrightarrow n \qquad (7\text{-}30)$$

7.3.3 Stringvergleiche

Strings können - ebenso wie andere Datentypen - miteinander verglichen werden. Dazu gibt es in FORTH spezielle Wörter. Mit diesen ist es unter anderem auch möglich, einen String zu durchsuchen, um herauszufinden, ob er einen bestimmten Teilstring enthält. Man möchte also herausfinden, ob eine bestimmte Zeichenfolge Teil dieses Strings ist. Besonders bei Programmen, die mit Textverarbeitung zu tun haben, ist dies eine nützliche Anwendung. Eine solche Suche bewirkt man mit dem Wort **INSTR.** Hier ein Anwendungsbeispiel:

```
ABS TOP INSTR                                        (7-31)
```

In diesem Fall wird der String an der Adresse **ABS** durchsucht, wobei FORTH versucht herauszufinden, ob der in **TOP** gespeicherte String darin enthalten ist. Wenn etwa der String in **TOP** das Wort **BUCH** enthält und der Wert von **ABS** lautet DAS BUCH IST AUF DEM TISCH, dann hinterläßt das Wort **INSTR** nach seinem Aufruf auf dem Stack die Integer 4. Dies bedeutet, daß der gesuchte Teilstring an der vierten Position im Zielstring zu finden ist. Sollte sich im Zielstring kein Vorkommnis des Suchstrings finden lassen, dann

pusht **INSTR** den Wert 0 auf den Stack. Wir haben folgende Stack-Relation:

$a_{\$1}$ $a_{\$2}$ --> n (7-32)

Oft ist es erforderlich, eine Anzahl von Strings in alphabetische Reihenfolge zu bringen. Jedem String kann man einen numerischen Wert zuordnen, der der Position entspricht, die dieser String in einer sortierten Liste haben würde. So hat z.B. der String "AAA" einen kleineren numerischen Wert als "AAB". Man sagt auch, daß der erste String in der Sortierfolge vor dem zweiten kommt. Entsprechend ist der numerische Wert von "AECDEDE" kleiner als der von "ZXE". Ebenso unterscheidet sich der Sortierwert (und der ASCII-Wert) von Kleinbuchstaben von dem der entsprechenden Großbuchstaben. Zum Vergleichen zweier Strings - wobei deren numerische Werte für den Vergleich herangezogen werden - benutzt man das Wort **$COMPARE**; hier ein Beispiel für seine Anwendung:

```
ABC TOP $COMPARE                                    (7-33)
```

Bei einem Aufruf von **$COMPARE** wird der numerische Wert des Strings, der an der Adresse **ABC** gespeichert ist, mit dem verglichen, den der String unter der Adresse **TOP** hat. Die beiden Adressen werden vom Stack entfernt, und - je nachdem, wie der Vergleich ausfällt - die Werte -1, 0 oder 1 werden auf den Stack gelegt. Die Zahl 0 signalisiert, daß die beiden Strings gleich sind. -1 drückt aus, daß der erste String (in diesem Fall **ABC**) in der Sortierordnung vor dem zweiten String (**TOP**) kommt. Im umgekehrten Fall legt **$COMPARE** den Wert 1 auf den Stack. Die Stack-Relation lautet:

$a_{\$1}$ $a_{\$2}$ --> n (7-34)

Als Beispiel für die String-Manipulation wollen wir jetzt ein FORTH-Programm schreiben, das eine Liste von Namen sortiert. Sie finden das Programm in Abbildung 7-5. Es benötigt zwei Blöcke. Wir gehen davon aus, daß Sie diese beiden Blöcke geladen haben. Wir haben Kommentare in das Programm eingestreut, um Ihnen das

Verständnis seiner Arbeitsweise zu erleichtern. Wenden wir uns jetzt den Einzelheiten dieses Programms zu. Es besteht aus insgesamt drei benutzerdefinierten FORTH-Wörtern. Mit dem ersten, das wir **ALPHAIN** getauft haben, geben wir die Daten ein. Das zweite Wort mit dem Namen **ALPHAOUT** besorgt die eigentliche Sortierarbeit und gibt dann das Ergebnis aus. Das dritte Wort, **ALPHA**, ruft lediglich diese beiden Wörter auf. In der ersten Zeile des ersten Blockes definieren wir zuerst einen String-Array mit dem Namen **NACHNAME**. Er umfaßt 11 Zeilen, von denen jede 20 Byte lang sein kann. Wir definieren des weiteren eine 21 Byte lange Stringvariable mit dem Namen **TEEM**. Weiterhin benötigt das Programm eine Stringkonstante, die den Namen **Q** trägt und aus einer Folge von 20 Z's besteht. (Wir gehen davon aus, daß der Benutzer die zu sortierenden Namen in Großbuchstaben eingibt.) Abschließend werden in Zeile 3 noch zwei Variablen mit dem Namen **NUMB** und **NUMB1** vereinbart, die zur Speicherung zweier einfach genauer Integers dienen.

Das Eingabewort **ALPHAIN** beginnt in Zeile 4 des ersten Blockes. In den Zeilen 5 und 7 wird der Array **NACHNAME** initialisiert. In jedem String dieses Arrays speichern wir im ersten Byte die Stringlänge (20). Die verbleibenden Stringelemente setzen wir auf 0. Dieser Schritt wäre unnötig, wenn das Programm immer nur einmal durchlaufen werden soll. In diesem Fall setzt FORTH ja automatisch den String-Array auf 0. Wir wollen das Wort jedoch mehrfach aufrufen; in diesem Fall befinden sich nach dem ersten Durchgang im Array immer noch die alten Sortierdaten, die für ein korrektes Funktionieren des Programms durch eben diesen Initialisierungsschritt erst entfernt werden müssen. Zeile 7 speichert im ersten Byte des Strings **TEMM** den Wert 20, vermerkt also ebenfalls die Stringlänge an dieser Stelle.

Die Dateneingabe wird in den Zeilen 9 bis 13 besorgt. Dies geschieht, indem das Programm vom Benutzer die Eingabe einer einfach genauen Integer verlangt. Diese entspricht der Anzahl von Namen, die sortiert werden sollen. Hier sollte der Benutzer keinen Wert angeben, der größer als 10 ist. (Genaugenommen könnten wir auch noch Zeile 0 unseres zweidimensionalen Arrays benutzen und dann 11 Namen sortieren; wir verzichten hier jedoch darauf.) Sollten Sie eine längere Namenliste sortieren lassen wollen, dann müssen Sie in Zeile 1 anstelle der 10 den gewünschten Wert eintragen. Die eigentlichen Sortierdaten werden in einer Schleife (Zeile 11 - 13) eingelesen und im Array **NACHNAME** gespeichert. Wir bedienen uns hierzu des Wortes **EXPECT**, da man mit diesem Wort

```
 0 ( Sortierprogramm  -  Block 1 von 2 )
 1 20 10  $ARRAY NACHNAME        20 $VARIABLE TEMM
 2      $CONSTANT Q ZZZZZZZZZZZZZZZZZZZZ"
 3      VARIABLE NUMB  VARIABLE NUMB1
 4 :ALPHAIN    (Variablen initialisieren und Daten eingeben)
 5          10 0 DO 20 I NACHNAME C!
 6           20 1 DO  0   J  NACHNAME  I +   C!   LOOP
 7      LOOP          20  TEMM C!
 8          (Anzahl der Namen eingeben)
 9      CR. " WIE VIELE NAMEN SOLLEN SORTIERT WERDEN " #IN NUMB !
10          (Liste der Namen eingeben)
11      CR NUMB@  1+  1    DO    ." WIE LAUTET DER NAME NR. " I .
12           CR     1  NACHNAME 1+   20 EXPECT      CR
13      LOOP  CR       ;
14
15
```

```
 0 (Sortierprogramm  -  Block 2 von 2)
 1 : ALPHAOUT (Sortieralgorithmus)
 2    (TEMM auf 20 mal "Z" setzen)
 3    NUMB @   1+   1  DO      Q    TEMM $!
 4        (Innere Schleife initialisieren und Vergleich starten)
 5        NUMB @ 1+  1  DO  I NACHNAME    TEMM    $COMPARE
 6            (Namen finden)
 7           0<  IF I NUMB1   !   I NACHNAME         TEMM $!
 8                  THEN LOOP
 9            (Namen ausgeben)
10              CR   NUMB1 @  NACHNAME  $,
11            (Ausgegebene Namen 'loeschen')
12           Q   NUMB1 @ LASTNAME  $!
13     LOOP ;
14
15 : ALPHA  ALPHAIN   ALPHAOUT   ;
```

ABBILDUNG 7-5: Ein Sortierprogramm

die Anzahl der Eingabezeichen automatisch auf 20 begrenzen kann. Dazu muß dem Wort **EXPECT** allerdings der Wert 20 auf dem Stack bereitgestellt werden (Zeile 12). Die auf den Stack gelegte Adresse ist **NACHNAME+1**. Dadurch wird die erste in den einzelnen Strings gespeicherte Zahl nicht verändert und behält den bei der Initi-

alisierung eingeschriebenen Wert 20 bei. In Zeile 13 endet die Definition von **ALPHAIN**.

Untersuchen wir jetzt den zweiten Block in der Abbildung 7-11. Hier findet der eigentliche Sortiervorgang statt. Wir bedienen uns dabei des folgenden Verfahrens: Wir gehen durch die gesamte Liste der zu sortierenden Namen und suchen den alphabetisch kleinsten Namen heraus. Diesen geben wir dann aus und löschen ihn aus der Liste. Anschließend gehen wir erneut die ganze Liste durch, suchen wieder nach dem kleinsten Namen, der jedoch jetzt, bezogen auf die Gesamtliste, der zweitkleinste Name sein muß, da ja der kleinste gelöscht wurde. Dieser wird ebenfalls ausgegeben und gelöscht. Der Prozeß wird so lange wiederholt, bis alle Namen aus der Liste gelöscht sind. Um den kleinsten Namen in der Liste zu finden, vergleichen wir jeden Eintrag mit einem Vergleichswert, der als erstes aus 20 Zs besteht. Ist der Listeneintrag kleiner als der Vergleichswert (und beim ersten Durchgang ist er dies mit Sicherheit immer), dann wird der Listenwert zum neuen Vergleichswert, und der zweite Listeneintrag wird mit dem neuen Vergleichswert verglichen. Ist er kleiner, dann ersetzt er den Vergleichswert, und wir gehen auf diese Art und Weise die gesamte Liste durch. Dadurch ist garantiert, daß wir das jeweils alphabetisch kleinste in der Liste enthaltene Element finden.

Das Wort, das all diese Operationen bewirkt, heißt **ALPHAOUT** und beginnt in Zeile 1. Es arbeitet mit zwei ineinander verschachtelten Schleifen. Die erste Schleife geht so oft durch die Liste, wie Einträge in ihr enthalten sind. Die zweite innere Schleife sucht bei jedem Durchgang das aktuelle kleinste Element heraus, gibt es aus und löscht es. Die äußere Schleife hat als Anfangswert 1 und als Testwert 1 plus die Anzahl der Namen im Array. In Zeile 3 setzen wir in der Variablen **TEMM** den anfänglichen Vergleichswert, der aus 20 Zs besteht und somit garantiert am Ende einer alphabetisch sortierten Liste zu stehen kommt. Jetzt geht es in die innere Schleife. **TEMM** wird mit jedem Nameneintrag in der Liste verglichen; falls der Listeneintrag kleiner als der Vergleichswert ist, dann ersetzt er den in **TEMM** gespeicherten Wert. Außerdem merken wir uns noch in der Variablen **NUMB1**, an welcher Position im Array der String zu finden ist, der als letzter den Vergleichswert in **TEMM** ersetzt hat. Nach einem vollständigen Durchgang durch die innerste Schleife enthält folglich **NUMB1** die Zeilennummer des alphabetisch kleinsten Namens in **NACHNAME**. Diesen Namen geben wir in Zeile 10 aus. Als nächstes wird der so gefundene Wert in Zeile 12 gelöscht, indem wir lauter Zs

in den String schreiben und ihn so an das Ende des Alphabets bringen. Es folgt ein neuer Durchgang durch die äußerste Schleife. Der im letzten Durchgang gefundene Name - der kleinste in der Gesamtliste - kommt jetzt nicht mehr als Kandidat in Frage, da er durch eine Folge von Zs ersetzt und somit an das Ende der alphabetischen Liste gebracht worden ist. Wenn wir jetzt zum zweiten Mal in die innerste Schleife eintreten, finden wir somit den zweiten Namen in der alphabetischen Reihenfolge (bezogen auf die Gesamtliste). Das Sortierwort **ALPHAOUT** endet in Zeile 13.

Ganz im Sinne der strukturierten Programmierung findet sich in Zeile 14 das Wort **ALPHA**, das lediglich die beiden Hilfswörter **ALPHAIN** und **ALPHAOUT** aufruft.

7.3.4 Weitere Stringfunktionen

Manchmal werden Strings - vor allen Dingen vom Ende her - mit Leerzeichen aufgefüllt. Beispielsweise könnten wir für einen String 64 Byte an Platz bereitgestellt haben, von denen aber nur 25 mit Zeichen beschrieben sind. Der verbleibende Platz wird mit Leerzeichen aufgefüllt. Dies kann zu gewissen Problemen führen. Wenn wir z.B. den String auf einem Drucker ausgeben wollen, so sollten die überflüssigen nachlaufenden Leerzeichen nicht mit ausge"druckt" werden. In FORTH ist es ja üblich, im ersten Byte eines Strings die Anzahl der darin gespeicherten Zeichen zu vermerken. Es wäre nun wünschenswert, diese Zahl zumindest zeitweilig durch eine zu ersetzen, die die Anzahl der tatsächlich gespeicherten Zeichen ohne nachlaufende Leerzeichen angibt. Diese Zahl könnte man dann verwenden, um Druckausgaben zu steuern. Das FORTH-Wort **-TRAILING** tut genau das Gewünschte: Es legt eine Zahl auf den Stack, die gleich der Stringlänge abzüglich nachlaufender Leerzeichen ist. Hier ein Beispiel:

```
ABC ABC -TRAILING                                  (7-35)
```

Bei dem Aufruf von **-TRAILING** wird die erste Adresse von **ABC** vom Stack entfernt und die berichtigte Zeichenzahl des Strings als einfach genaue Integer auf den Stack gelegt. Jetzt enthält der Stack also die Anfangsadresse von **ABC** und die "Nettozeichenzahl".

Letztere ist an oberster Stack-Position. Beachten Sie, daß **-TRAILING** am erste Byte von **ABC** keine Veränderungen vornimmt.

Es ist auch möglich, nachlaufende Leerzeichen aus einem String zu löschen. Dazu bedient man sich des Wortes **$-TB**. Wenn wir z.B. folgende Wörter ausführen:

```
ABC $-TB
```

dann wird das erste in **ABC** gespeicherte Byte so verändert, daß es die tatsächliche Zeichenzahl in **ABC** ohne nachlaufende Leerzeichen wiedergibt.

7.3.5 Numerische Stringinformationen

Es gibt in FORTH auch Wörter, mit denen man sich numerische Informationen über Strings beschaffen kann. Diese wollen wir jetzt besprechen. Mit dem Wort **LEN** erfährt man die Länge eines Strings. Durch Ausführung von

ABC LEN (7-36)

wird die Adresse von **ABC** vom Stack entfernt und durch eine einfach genaue Integer ersetzt, die der Anzahl der Zeichen in dem String entspricht.

Das FORTH-Wort **ASC** liefert den ASCII-Code des ersten Zeichens in einem String. Das Beispiel

ABC ASC (7-37)

sorgt dafür, daß die Adresse von **ABC** vom Stack entfernt und an ihrer Stelle dort der ASCII-Code des ersten Zeichens im String abgelegt wird. Die beiden Wörter **LEN** und **ASC** haben die Stack-Relation

a --> n (7-38)

Das FORTH-Wort **CHR$** erwartet eine einfach genaue Integer auf dem Stack (diese muß einem gültigen ASCII-Code entsprechen) und ersetzt diese durch die entsprechende Zeichendarstellung im temporären Arbeitsbereich. Um auf das Ergebnis zugreifen zu können, legt **CHR$** die Adresse dieses Arbeitsbereiches (**PAD**) als Ergebnis auf den Stack. Wir haben folgende Stack-Relation:

$n \text{ --> } a_{pad}$ (7-40)

Oft ist es erforderlich, in Stringdarstellung gespeicherte Zahlen in ihre numerischen Entsprechungen umzuwandeln. Ein String, der aus lauter Ziffern besteht, stellt zwar - für den Menschen - eine Zahl dar, der Computer kann jedoch damit nicht rechnen. Dazu muß der String zuvor umgewandelt werden, was das FORTH-Wort **VAL** besorgt. Dieses wandelt eine als String gegebene Zahl in die entsprechende einfach genaue Integer um. Wenn also beispielsweise in **ABC** der String "123" gespeichert ist, dann haben wir nach Ausführung von

ABC VAL

auf dem Stack die einfach genaue Integer 123 liegen. Die Stack-Relation lautet:

a --> n (7-41)

7.4 Übungsaufgaben

Wenn in den folgenden Aufgaben von Ihnen verlangt wird, FORTH-Programme zu schreiben, dann überprüfen Sie diese am besten an Ihrem Computer. Halten Sie die einzelnen FORTH-Wörter so kurz wie möglich; versuchen Sie modular zu programmieren.

7-1 Was ist der Unterschied zwischen einem Zeichen und einem String?

7-2 Vergleichen Sie die FORTH-Wörter **C!** und **C@** mit **!** und @.

7-3 Richten Sie zwei Arrays ein. Speichern Sie im ersten Integers, und duplizieren Sie diese mittels **CMOVE** in einen zweiten Array. Dazu muß der zweite Array natürlich mindestens so groß wie der erste sein.

7-4 Wiederholen Sie Aufgabe 7-3 unter Verwendung von **MOVE**.

7-5 Warum kann man in Aufgabe 7-3 sowohl mit **CMOVE** als auch mit **<CMOVE** arbeiten?

7-6 Schreiben Sie ein FORTH-Wort, das Zeichen von der Tastatur einliest und ihren ASCII-Code auf dem Bildschirm ausgibt. Das Programm sollte durch Eingabe eines Prozentzeichens beendet werden.

7-7 Schreiben Sie ein Programm, mit dem Sie Text eingeben und in einen Array speichern können. Lassen Sie sich dann Ihre Eingabe satzweise ausgeben. Nach Ausgabe eines Satzes sollte das Programm so lange warten, bis Sie einen neuen eingeben.

7-8 Schreiben Sie das Programm der Abbildung 7-1 neu, so daß Sie Zeichen im gespeicherten Text verändern können.

7-9 Schreiben Sie das Programm der Abbildung 7-1 neu, so daß Sie Zeichen aus dem Text löschen können.

7-10 Schreiben Sie das Programm der Abbildung 7-2 neu, so daß nicht zu viele Zeichen in den Array **TEXT** mitaufgenommen werden können.

7-11 Schreiben Sie das Programm in Abbildung 7-1 neu, so daß es mit dem FORTH-Wort **EXPECT** arbeitet. Beenden Sie Ihre Eingaben durch Drücken der Return-Taste.

7-12 Schreiben Sie mittels **WORD** ein FORTH-Programm, das einen String durchgeht und das erste Wort darin ausgibt. Dann hält das Programm an und wartet so lange, bis Sie ein beliebiges Zeichen eingeben, woraufhin das nächste Wort ausgegeben

wird. Dieser Prozeß soll so lange fortgesetzt werden, bis alle Wörter im String ausgegeben sind.

7-13 Wiederholen Sie Aufgabe 7-12, wobei Sie diesmal mit dem Trennzeichen "#" arbeiten.

7-14 Was ist der Unterschied zwischen Stringkonstanten und numerischen Konstanten?

7-15 Ändern Sie das Programm der Abbildung 7-1 so, daß es mit den Wörtern aus Abschnitt 7-3 arbeitet.

7-16 Wiederholen Sie Aufgabe 7-8 mit den Wörtern aus Abschnitt 7-3.

7-17 Wiederholen Sie Aufgabe 7-9 mit den Wörtern aus Abschnitt 7-3.

7-18 Wiederholen Sie Aufgabe 7-10 mit den Wörtern aus Abschnitt 7-3.

7-19 Ändern Sie das Programm aus der Abbildung 7-5 so, daß der Originalarray erhalten bleibt.

7-20 Wiederholen Sie die Aufgabe 6-16, wobei diesmal der Verkäufername mitaufgenommen werden soll. Setzen Sie hier jedoch nicht die Wörter aus Abschnitt 7-3 ein.

7-21 Wiederholen Sie Aufgabe 7-20 mit den Wörtern aus Abschnitt 7-3.

7-22 Wiederholen Sie Aufgabe 7-21, lassen Sie sich diesmal jedoch die Werte nach den Verkäufernamen alphabetisch sortiert ausgeben.

7-23 Erörtern Sie die Arbeitsweise von **$+**.

7-24 Warum sollten Sie nicht über längere Zeit Daten im temporären Arbeitsspeicher zwischenspeichern?

7-25 Ändern Sie das Programm der Abbildung 7-5 so ab, daß nur die ersten vier Zeichen eines jeden Namens für den Vergleich herangezogen werden.

7-26 Wiederholen Sie das Programm der Abbildung 7-1 so, daß Sie die gespeicherten Daten nach einem bestimmten Teilstring durchsuchen können.

7-27 Wiederholen Sie Aufgabe 7-26 ohne das FORTH-Wort **INSTR.** Hinweis: schreiben Sie Ihr eigenes Wort, das diese Funktion erledigt.

7-28 Vergleichen Sie die Wörter **-TRAILING** und **$-TB.**

7-29 Schreiben Sie ein Programm, das zwei als Strings gespeicherte Zahlen addiert.

8

Diskettenoperationen

8 Diskettenoperationen

In diesem Kapitel gehen wir die Prinzipien der Disketten-Ein- und -Ausgabe durch. Einige Informationen darüber haben wir schon in Kapitel 1 besprochen; hier haben Sie erfahren, wie Sie Ihre Programme auf Diskette speichern und später laufen lassen können. Dieses Thema wollen wir im vorliegenden Kapitel noch einmal aufgreifen. Darüber hinaus beschäftigen wir uns mit der wichtigen Frage der Datenorganisation auf Disketten. Viele der dabei behandelten Themen hängen eng miteinander zusammen. Etliche Techniken zum Speichern von Programmen können auch für die Datenspeicherung eingesetzt werden. Einige Details, die wir hier darstellen, können von System zu System etwas unterschiedlich ausfallen.

8.1 Prinzipien der Datenspeicherung auf Diskette

Dieses Kapitel hat zum Ziel, die in Kapitel 1-4 eingeführten Kenntnisse über die Diskettenspeicherung zu erweitern und zu vertiefen. Deshalb wiederholen wir etwas von dem Material, das wir im ersten Kapitel bereits eingeführt haben.

Reine FORTH-Systeme werden mit einem eigenen Betriebssystem ausgeliefert. Verglichen mit anderen Betriebssystemen sind diese jedoch sehr einfach strukturiert. Ein Vorteil der ausschließlichen Arbeit in FORTH besteht jedoch darin, daß Sie dieses einfache Betriebssystem leicht modifizieren und Ihren persönlichen Zwecken anpassen können. So verwalten die meisten Betriebssysteme Disketteninhaltsverzeichnisse mit einer äußerst komplexen Struktur; im Vergleich dazu ist das vom FORTH-Betriebssystem verwaltete Inhaltsverzeichnis rudimentär. Sie können es als Benutzer jedoch leicht modifizieren und Ihren eigenen Bedürfnissen anpassen. Andere FORTH-Systeme verwalten die benötigten Informationen zwar im Inhaltsverzeichnis, tun dies jedoch nicht in einer Ihnen genehmen Form; auch diese Systeme können meistens nach den Benutzerwünschen modifiziert werden. Wieder andere FORTH-Systeme verwenden das eigentliche Betriebssystem des Computers, enthalten aber ihre eigenen Kommandos für Disketten-Ein- und -Ausgabe.

Um die nachfolgenden Erörterungen besser verstehen zu können, wollen wir erst einmal darstellen, wie Programme auf Disketten

gespeichert werden. Wie Sie bereits aus Kapitel 1-4 wissen, werden die meisten FORTH-Programme in einem FORTH-Editor eingetippt und anschließend auf Diskette gespeichert. Das FORTH-System arbeitet dabei mit sog. Datenblöcken, von denen jeder genau 1024 Byte fassen kann. (Genaugenommen kann diese Zahl von System zu System variieren; bei 1024 handelt es sich jedoch um einen typischen Wert.) Stellen Sie Sich jetzt einmal vor, Sie geben ein Programm mit Ihrem Editor ein. Die Zeichen, die Sie auf Ihrer Tastatur tippen, werden nicht direkt auf Diskette gespeichert, sondern in einem speziell dafür reservierten Bereich im Arbeitsspeicher abgelegt. Diesen Bereich bezeichnet man als Blockpuffer. Wie Sie wissen, ist ein Puffer ein Speicherbereich, der für die temporäre Speicherung von Daten vorgesehen ist. In FORTH ist der Pufferbereich seinerseits wieder in Blöcke unterteilt, die der Blockgröße auf Diskette - in unserem Beispiel 1024 - entsprechen. Blöcke aus dem Arbeitsspeicherpuffer können auch auf Diskette gespeichert werden. Man spricht dann von Diskettenblöcken. Ihr FORTH-System reserviert in der Regel eine ganze Anzahl solcher Blockpuffer im Arbeitsspeicher. Jeder von der Diskette geladene oder auf Diskette zu schreibende Block wird in einem Puffer abgelegt. Bei einigen Systemen ist die Anzahl der Puffer festgelegt, bei anderen kann sie variieren. Sehen Sie also in Ihrem Handbuch nach, um die Usancen Ihres Systems zu erfahren.

Sie haben also nun ein Programm eingegeben und in einem Blockpuffer gespeichert. Die meisten Editoren verfügen über Kommandos, mit denen Sie den Pufferblock markieren können, so daß er bei der nächsten Aktualisierungsoperation auf Diskette gespeichert wird; meistens bedient man sich dazu des FORTH-Wortes **UPDATE**. Gelegentlich arbeiten Sie mit langen Programmen, die nicht in einem einzigen Block Platz finden. Es kann sogar sein, daß Ihr Programm mehr Blöcke belegt, als im Pufferbereich reserviert sind. Der Einfachheit halber wollen wir davon ausgehen, daß dieses hypothetische Programm in drei Diskettenblöcke gespeichert ist, Ihr FORTH-System im Arbeitsspeicher jedoch nur zwei Pufferblöcke bereitstellt. Zur Bearbeitung von drei Blöcke ist es nötig, einen der Speicherblöcke zu überschreiben. Wenn Sie einen Pufferblock mittels **UPDATE** markieren und nachfolgend versuchen, diesen zu überschreiben, dann wird der Block automatisch auf die Diskette zurückgeschrieben, ehe der Inhalt überschrieben werden kann. Das bedeutet, daß das FORTH-System Blöcke schützt, die zur Aktualisierung markiert wurden. (Vergewissern Sie sich anhand Ihres FORTH-Handbuches, daß dies in Ihrem System auch wirklich so ist.)

Wie wir gesehen haben, können Sie Ihr eingegebenes Material mittels **UPDATE** markieren und so vor Verlust schützen. Auch wenn Sie den Editor versehentlich verlassen, werden die markierten Blökke auf Diskette gespeichert. Darüber hinaus erlauben viele Systeme dem Benutzer auch, den Editor ohne Sicherung zu verlassen, indem etwa die BREAK-Taste gedrückt wird.

Wir können auch selbst dafür sorgen, daß in Blockpuffern enthaltenes Material auf Diskette geschrieben wird. Ehe wir das Wort betrachten, mit dem man das bewirkt, wollen wir noch einige Details des Ediervorgangs untersuchen. Die Blöcke auf einer Diskette sind durchnumeriert. (Die Methode, nach der diese Numerierung geschieht, wollen wir noch später besprechen.) Sie müssen dem Editor beim Aufruf die Nummer des Blockes mitteilen, die er bearbeiten soll. In der Regel ruft man den Editor über das FORTH-Wort **EDIT** auf. Zur Bearbeitung des Blöckes 123 müssen Sie also eingeben:

123 EDIT (RETURN) (8-1)

EDIT entfernt also eine Integer vom Stack und liest den entsprechenden Block von der Diskette in einen Pufferblock. Anschließend befinden Sie sich im Editor und können diesen Block mit den entsprechenden Kommandos bearbeiten. **EDIT** hat die Stack-Relation

n --> (8-2)

Eine spezielle FORTH-Variable mit dem Namen **SCR** speichert stets die Nummer des Pufferblockes, der gerade bearbeitet wird. Weiterhin merkt sich das System die Nummer des Diskettenblockes, der die Daten enthält, die im in Arbeit befindlichen Pufferblock gespeichert sind. Beachten Sie, daß zur Speicherung dieses Wertes **SCR** nicht herangezogen werden kann, da Sie ja mit mehreren Blökken gleichzeitig arbeiten können. Die in **SCR** enthaltene Zahl ist stets die Blocknummer des letzten bearbeiteten Blockes. Angenommen, Sie haben einen oder mehrere Blöcke bearbeitet und für die Aktualisierung markiert. Wenn Sie jetzt das FORTH-Kommando **SAVE-BUFFERS** eingeben, dann wird jeder markierte Speicherpuffer an die passende Stelle auf der Diskette gesichert. Viele FORTH-Systeme benutzen auch anstelle von **SAVE-BUFFERS** das Kommando **FLUSH**.

Gelegentlich arbeiten Sie mit einem im Teststadium befindlichen Programm und wollen nicht, daß die gerade im Blockpuffer befindliche Version die auf der Diskette gespeicherte Version überschreibt. Dabei ist zu beachten, daß das Zurückschreiben unbenutzter Blöcke vom FORTH-System manchmal automatisch ausgeführt wird. Um so etwas zu verhindern, können Sie mit dem Wort **EMPTY-BUFFERS** einen Puffer vor dem Zurückschreiben auf die Diskette bewahren. Die Ausführung von **EMPTY-BUFFERS** sorgt nämlich dafür, daß die Aktualisierungsmarken von allen Speicherpuffern entfernt werden. **EMPTY-BUFFERS** ändert nichts an dem Material, das in den Blockpuffern gespeichert ist. Sie können dieses also weiterhin mit dem Editor bearbeiten. Wenn Sie es wollen, können Sie dann ohne weiteres im Anschluß daran einen Puffer wieder mittels **UPDATE** markieren und so dafür sorgen, daß er bei der nächsten Aktualisierung berücksichtigt wird. Denken Sie jedoch daran, daß **UPDATE** stets nur den Block markiert, der als letzter in Arbeit befindlich war.

Beim Edieren ist es gelegentlich erforderlich, sich das Wort anzuschauen, das man gerade bearbeitet. Zwei FORTH-Wörter können dazu eingesetzt werden. Eines davon lautet **L**. Um es auszuführen geben Sie einfach ein:

L (RETURN) (8-3)

Der aktuelle Block - also der, dessen Nummer in **SCR** gespeichert ist - wird dann auf dem Bildschirm ausgegeben. Um auch noch zusätzlich eine Druckausgabe zu erreichen, verwenden Sie das Wort **PCRT**. Mit **L** können Sie jedoch nur den aktuellen Block auflisten. Wenn Sie einen anderen als den gerade in Arbeit befindlichen Block sehen wollen, dann müssen Sie das Kommando **LIST** geben. Um den Block 123 aufzulisten, müssen Sie also eingeben:

123 LIST (RETURN) (8-4)

Daraufhin wird die Zahl 123 vom Stack entfernt und in **SCR** gespeichert. Anschließend wird der betreffende Block ausgegeben. Die Stack-Relation für **LIST** finden Sie ebenfalls in (8-2).

Wenn **SCR** sowieso schon den richtigen Wert enthält, dann brauchen Sie nicht das Wort **EDIT** einzugeben. Im MMSFORTH reicht es, das Kommando **E** zu tippen. Dieses bewirkt, daß der Block bearbeitet wird, dessen Nummer in **SCR** gespeichert ist.

Nun ein paar Wörter zur Compilierung von FORTH-Programmen. Dieser Vorgang muß stattfinden, ehe Sie ein Programm ausführen können. "Compilieren" bedeutet, daß unter anderem Wörterbucheinträge für Wörter und Variable und Verweise auf den Maschinencode angelegt werden, der die entsprechenden Operationen auslöst. Zum Compilieren eines Blockes geben Sie das FORTH-Kommando **LOAD.** Daraufhin wird der Block geladen, und alle darin enthaltenen Wörter werden ausgeführt. Um das Programm zu compilieren, das in Block 123 gespeichert ist, müssen Sie also eintippen:

123 LOAD (RETURN) (8-5)

In diesem Fall wird die Zahl 123 vom Stack entfernt und der so angegebene Block geladen. Sind in dem Block Definitionen mittels **:** und **;** enthalten, so werden die entsprechenden Wörter compiliert; direkt ausführbare Anweisungen wie etwa

VARIABLE ABC

werden hingegen sofort ausgeführt. Die Relation (8-2) gilt ebenfalls für **LOAD.**

Gelegentlich ist es nötig, mehrere Blöcke auf einmal zu laden. In diesem Fall können Sie mit dem MMSFORTH-Kommando **LOADS** arbeiten. Dieses Wort ist kein Teil des FORTH-79-Standards. Um Block 123 und die nachfolgenden beiden Blöcke zu laden, müssen Sie eingeben:

123 3 LOADS (RETURN) (8-6)

Bei **LOADS** lautet die Stack-Relation folgendermaßen:

n_1 n_2 --> (8-7)

In einigen FORTH-Systemen ist **LOADS** anders definiert und funktioniert entsprechend anders. Überprüfen Sie deshalb vorher die Informationen in Ihrem FORTH-Handbuch.

Eine andere Methode zum Laden mehrerer Blöcke besteht darin, als letzte Anweisung in einem Block das FORTH-Kommando aufzunehmen, das den nächsten Block lädt. Wenn nun der erste Block geladen wird, dann wird diese Anweisung ausgeführt und sorgt dafür, daß auch der nächste Block nachgeladen ist. Zum Laden der beiden Blöcke 123 und 124 nehmen Sie also in Ihren Block 123 als letzte Anweisung auf:

```
124 LOAD
```

Wenn Sie jetzt den Block 123 laden, dann hat dies zur Folge, daß auch Block 124 automatisch nachgeladen wird. Diesen Prozeß kann man wiederholen und so eine beliebige Anzahl von Blöcken miteinander verketten. Wenn sich ein einzelnes FORTH-Wort allerdings über mehr als einen Block erstreckt, dann ist dieses Verfahren nicht anwendbar. Die einfachste Lösung für dieses Problem besteht darin, kürzere Wörter zu schreiben, die in einem Block Platz finden. Strukturieren Sie Ihr Problem neu, und schreiben Sie kurze FORTH-Wörter, die ihrerseits andere Wörter rufen.

Die meisten FORTH-Systeme verwalten keine Inhaltsverzeichnisse für Disketten. In manchem System findet sich jedoch ein Wort, mit dem man herausfinden kann, welche Information in den einzelnen Blöcken gespeichert sind; es lautet **INDEX**. Hier ein Beispiel für seine Anwendung:

```
50 20 INDEX (RETURN)                                        (8-8)
```

Damit sorgen wir dafür, daß jeweils die erste Zeile von 20 Blökken aufgelistet wird, wobei dieser Vorgang mit Block 50 beginnt. Da es guter Programmierstil ist, in jedem Block am Anfang einen Kommentar zu schreiben, der die Funktion der darin enthaltenen Anweisungen schildert, kann man mit **INDEX** Aufschluß über die auf der Diskette gespeicherten Blöcke erhalten. Außerdem können Sie so ausfindig machen, welche Diskettenblöcke leer sind und für die Speicherung neuer Programme herangezogen werden können. Hier lautet die Stack-Relation folgendermaßen:

n_1 n_2 --> (8-9)

Mit den Diskettenblöcken sind wir jetzt vertraut; es wird also Zeit, sich die Organisation und Informationen auf Diskette anzusehen und zu erfahren, nach welcher Methode Blocknummern vergeben werden. Wir beschränken uns hier auf Informationen über die Diskettenspeicherung, die in unserem Zusammenhang relevant sind. Die angegebenen Zahlen sind typische Werte für kleine Diskettensysteme. Die dargestellten Prinzipien gelten jedoch auch für größere Systeme und können auf diese übertragen werden. Daten werden auf Disketten in sog. Spuren gespeichert. Einige Laufwerke arbeiten mit 35 Spuren, während andere 40 oder sogar 80 Spuren aufweisen. Für unsere Erörterungen wollen wir annehmen, daß die Laufwerke des Systems über 40 Spuren verfügen. Die Spuren einer Diskette sind als konzentrische Ringe über den Datenträger verteilt und ihrerseits wieder in kleinere Einheiten, die sog. Sektoren, unterteilt. Die Anzahl Sektoren pro Spur wird von der Aufzeichnungstechnik bestimmt, mit der das jeweilige System arbeitet. Wir gehen davon aus, daß unser hypothetisches Laufwerksystem 10 Sektoren pro Spur schreibt. Typischerweise können in einem Diskettensektor 256 Byte gespeichert werden. Der Sektor ist die kleinste logische Einheit auf einer Diskette. Um einen Block in FORTH zu speichern, werden somit 4 Sektoren benötigt. Auf den meisten Diskettensystemen sind einige Diskettensektoren reserviert, da in ihnen ein Urladeprogramm für das "Hochfahren" des Systems gespeichert ist. Die Einzelheiten dieses Programms brauchen uns hier nicht zu beschäftigen; wir gehen lediglich davon aus, daß unser hypothetisches System zwei Sektoren für den Urlader reserviert.

Fassen wir noch einmal zusammen: Wir stellen uns ein Diskettensystem mit 40 Spuren zu 10 Sektoren vor, so daß auf einer Diskette 99 Blöcke gespeichert werden können. Die Diskette verfügt über insgesamt 400 Sektoren. Zwei Sektoren gehen für den Urlader ab, so daß auf der Diskette insgesamt 398 Sektoren verwendet werden (denn 99 Blöcke belegen 99 x 4 = 396 Sektoren). Die verbleibenden beiden Sektoren auf der Diskette bleiben ungenutzt. Jetzt nehmen wir weiter an, daß wir mit mehr als einem Laufwerk arbeiten. Die meisten FORTH-Systeme vergeben fortlaufende Blocknummern. Es können sich nun zwei Disketten, eine in Laufwerk 0 (oder A) und eine in Laufwerk 1 (oder B) befinden. (Beachten Sie, daß der erste Sektor in der ersten Spur vom ersten Laufwerk die Nummer 0 erhält.) Der Block mit der Nummer 98 ist der letzte Block, der

auf der Diskette in Laufwerk 0 (oder A) gespeichert werden kann. Entsprechend wird der Block mit der Nummer 99 der erste Block auf Laufwerk 1 (oder B) sein. Die Nummer eines Blockes hängt also auch davon ab, auf welchem Laufwerk er sich befindet. Angenommen, ein bestimmter Block ist der 61ste auf seiner Diskette. Wenn Sie diese Diskette nun in Laufwerk 0 legen, dann erhält der fragliche Block die Nummer 60. Sollte sich hingegen die Diskette im zweiten Laufwerk (Laufwerk 1) befinden, dann ist die Blocknummer für diesen Block gleich 159. Da die Blöcke bei 0 beginnend durchnumeriert werden, trägt der 61ste Block die Blocknummer 60.

Wir wollen die eben besprochenen Informationen an einem kleinen Programm vertiefen, das ein Disketteninhaltsverzeichnis ausgibt. Wir wollen annehmen, daß auf jeder Diskette eine Kopie dieses Programms vorhanden ist. Es wäre z.B. günstig, das Programm auf jeder Diskette im 61sten Block (Nummer 60) abzuspeichern. Ist die Diskette im Laufwerk 0, dann können wir das Programm laden, indem wir uns den Block 60 holen; ist sie in Laufwerk 1, so müssen wir Block 159 ansprechen. Nach Laden des Programms können Sie sich ein Inhaltsverzeichnis der Programme auf Ihrer Diskette anzeigen lassen, indem Sie den Befehl **DIRECTORY** eingeben. Sie sehen dann auf Ihrem Bildschirm die Namen der einzelnen Programme, die sich auf der Diskette befinden. Um eines dieser Programme zu laden, *geben Sie dann lediglich den Namen des Programms* ein und drücken die Return-Taste. Das Wort sorgt dann selbst dafür, daß das von Ihnen gewählte Programm geladen und die Anfangsblocknummer in der Variablen **SCR** abgelegt wird; Sie können es somit edieren und auflisten.

Wenden Sie Sich nun der Abbildung 8-1 zu, in der Sie ein Listing des fraglichen Programms sehen. In Zeile 1 wird ein Wort **DRIVNO** definiert, das den Benutzer fragt, in welchem Laufwerk sich die Diskette befindet. Der Benutzer antwortet mit einer Zahl, die von dem Wort **DRIVNO** auf den Stack gelegt wird. In unserem Beispielprogramm nehmen wir an, daß die Diskette drei Programme enthält, wobei das Programm mit dem Namen **FACTORIAL** im Block Nummer 20 gespeichert ist, **FACTREAL** im Block 21 abgelegt und **ALPHA** im Block 46 und 47 zu finden ist. In Zeile 2 wird nun zuerst das Wort **FACTORIAL** definiert. Es ruft als erstes **DRVNO** und besorgt sich so auf dem Stack die Nummer des richtigen Laufwerks. Aus dieser Nummer erzeugen wir durch Multiplikation und Addition die Nummer des gewünschten Blockes, welche wir duplizieren. Die erste Kopie der Blocknummer speichern wir in der Variablen **SCR**, wodurch die Möglichkeit zum Bearbeiten des Programms gesichert ist; mit der

```
 0 ( Ein einfaches Disketten-Inhaltsverzeichnis )
 1 : DRVNO   ." BITTE LAUFWERKSNUMMER EINGEBEN "   #IN    ;
 2 : FACTORIAL DRVNO  99   *   20  +  DUP  SCR !  LOAD  ;
 3 : FACTREAL  DRVNO  99   *   21  +  DUP  SCR !  LOAD  ;
 4 : ALPHA     DRVNO 99  *  46  +  DUP  SCR  !  DUP
 5           LOAD   1+    LOAD  ;
 6
 7
 8
 9
10 : DIRECTORY CR ." FACTORIAL      FACTREAL      ALPHA    " ;
11
```

ABBILDUNG 8-1: Ein einfaches Inhaltsverzeichnis

zweiten Kopie laden wir den Block, der die eigentliche Programmdefinition enthält. Wir haben so das gewünschte Ergebnis erzielt. Das Wort in Zeile 3 der Abbildung 8-1 führt im wesentlichen dieselben Verarbeitungsschritte aus. Zeile 4 zeigt die Definition des FORTH-Wortes **ALPHA.** Nun haben wir ja vereinbart, daß das Wort **ALPHA** in zwei Blöcke gespeichert ist; deshalb muß die Definition der Zeile 4 auch dafür sorgen, daß zwei Blöcke geladen werden. **ALPHA** unterscheidet sich von den beiden anderen Wörtern hauptsächlich dadurch, daß vor dem ersten Ladevorgang die Blocknummer dupliziert wird; nach Abschluß des Ladens steht sie also noch auf dem Stack, kann um 1 erhöht und für den nächsten Ladevorgang herangezogen werden. **ALPHA** sorgt so dafür, daß die beiden richtigen Blöcke geladen werden und zur Editierung bereitstehen.

In Zeile 10 sehen Sie die Definition des Wortes, mit dem das Inhaltsverzeichnis aufgelistet werden kann; es heißt **DIRECTORY.** Dieses Wort tut nichts anderes, als die Namen aller Programme auszugeben. Unser Inhaltsverzeichnisprogramm funktioniert also wie gewünscht, wenngleich es noch sehr primitiv ist. Jedesmal, wenn Sie ein neues Programm auf Ihrer Diskette abspeichern wollen, dann müssen Sie das Programm für das Inhaltsverzeichnis auch ändern. Wenn wir ein "ausgewachsenes" Betriebssystem schreiben würden, dann müßten all diese Aufgaben automatisch vom System erledigt werden.

Zwei weitere FORTH-Kommandos haben mit der Verarbeitung von Blökken und Blockpuffern zu tun. Das erste lautet **BLOCK.** Hier ein typisches Anwendungsbeispiel für dieses Wort:

132 BLOCK (8-10)

In diesem Fall wird die Zahl 123 vom Stack entfernt und dann der Block mit der Nummer 123 von den entsprechenden vier Diskettensektoren in einen Blockpuffer im Arbeitsspeicher geladen. Als Resultat legt **BLOCK** weiterhin noch die Anfangsadresse des Puffers auf den Stack. Diese Adresse benötigt das FORTH-System, um den Puffer verwalten zu können. Auch Sie können sie dazu benutzen, um Daten zu speichern. Wir haben für **BLOCK** folgende Stack-Relation:

n --> a (8-11)

Ein weiteres, im Zusammenhang mit Ein-/Ausgaben benötigtes Wort, lautet **BUFFER**. Hier ein Anwendungsbeispiel:

123 BUFFER (8-12)

Dieses Beispiel sorgt dafür, daß die Zahl 123 vom Stack entfernt und daraufhin 1024 Byte aus dem Arbeitsspeicher als Arbeitspuffer für Block 123 zugewiesen werden. Die Anfangsadresse dieses Blockpuffers wird auf den Stack gelegt. Das Wort **BUFFER** arbeitet also ähnlich wie **BLOCK**. Der Unterschied besteht darin, daß ein Aufruf von **BUFFER** noch keine Information von der Diskette liest. **BUFFER** dient somit nur dazu, einen Speicherpuffer für nachfolgende Diskettenoperationen zu reservieren. Die Stack-Relation für **BUFFER** sehen Sie ebenfalls in (8-11).

8.2 Datenorganisation auf Disketten

Wir werden jetzt darlegen, wie Daten auf Diskette gespeichert und dort wiedergefunden werden. Während der Ausführung eines Programms kommen oft Diskettenzugriffe vor. Einige der hierfür in unserem Buch vorgestellten Wörter finden sich nicht in FORTH-79. Sie sind jedoch Bestandteil von MMSFORTH; weiterhin verfügen viele andere FORTH-Systeme über ähnliche Befehle.

DWTSECS und **DRDSECS** - Mit diesen beiden Wörtern kann man in FORTH Informationen auf die Diskette schreiben bzw. davon lesen. Sie sind kein Bestandteil von FORTH-79, lassen sich jedoch sehr bequem handhaben.

Mit dem Wort **DWTSECS** kann man Daten auf Diskette schreiben. Hier ein typisches Anwendungsbeispiel:

```
NUMB 1 23 8 2 DWTSECS
```
(8-13)

Bei diesem Beispiel nehmen wir an, daß **NUMB** eine Array-Variable ist. Das Beispiel (8-13) sorgt dafür, daß die Information im Arbeitsspeicher, die bei der Adresse von **NUMB** beginnt, auf die Diskette geschrieben wird. Die Daten werden auf der Diskette im Laufwerk 1 gespeichert. Insgesamt zwei Sektoren werden geschrieben, wobei mit Sektor 8 der Spur 23 begonnen wird. (Denken sie daran, daß die Numerierung von Laufwerken, Spuren und Sektoren jeweils mit 0 beginnt.) Da ein Sektor 256 Byte aufnehmen kann, bedeutet dies, daß 512 Byte aus dem Arbeitsspeicher gelesen und auf Diskette gespeichert werden. Wenn Sie sich des Wortes **DWTSECS** bedienen, dann spielt es keine Rolle, welche Art von Informationen im Speicher befindlich ist; das Wort schreibt einfach eine zusammenhängende Folge von Bytes auf Diskette, ohne sich um den Datentyp der so gespeicherten Information zu kümmern. So braucht **NUMB** beispielsweise nicht unbedingt ein Array zu sein; ebensogut könnte es sich dabei um eine Variable handeln, in der eine einfach genaue Integer gespeichert ist. Trotzdem werden bei Ausführung von (8-13) 512 aufeinanderfolgende Bytes, beginnend bei der Adresse **NUMB**, auf Diskette kopiert. Das Wort hat folgende Stack-Relation:

$$a\ n_1\ n_2\ n_3\ n_4\ \text{-->}\ n_{flag} \qquad (8\text{-}14)$$

Hierbei steht n_1 für die Laufwerksnummer, n_2 für die Spur, n_3 für die Sektornummer auf der Spur und n_4 für die Anzahl der Sektoren, die übertragen werden sollen. Das Wort hinterläßt ein Flag auf dem Stack. Damit wird dem Benutzer oder anderen Wörtern signalisiert, ob die Kopieroperation erfolgreich war. Bei der Ein-/Ausgabe von Daten auf Disketten können nämlich Schreib- oder Lesefehler auftreten. Das FORTH-System überprüft zuerst, ob die

Datenübertragung fehlerfrei vonstatten gegangen ist und legt in diesem Fall eine 0 auf den Stack. Wie bereits bei anderen Beispielen steht diese für den Wahrheitswert "falsch". Sollten bei der Datenübertragung Fehler aufgetreten sein, so legt **DWTSECS** hingegen den Wahrheitswert "wahr" als Flag auf den Stack. Sie können in einem Programm dieses Flag dazu benutzen, die Ein-/Ausgabeoperationen abzubrechen, zu wiederholen oder irgendwelche anderen Aktionen einzuleiten, die Ihnen sinnvoll erscheinen.

Den umgekehrten Vorgang, nämlich das Lesen von Daten von Diskette in den Arbeitsspeicher, löst man mit **DRDSECS** aus. In seiner Arbeitsweise ähnelt es sehr **DWTSECS**. So bewirkt z.B.

```
NUMB1 1 23 8 2 DRDSECS                    (8-15)
```

daß die Information, die sich im Laufwerk 1 auf Spur 23 in den Sektoren 8 bis 9 befindet, in den Arbeitsspeicher übertragen und dabei mit der Adresse **NUMB** begonnen wird. Führen wir nacheinander zuerst (8-13) und dann (8-15) aus, dann werden 512 Datenbyte aus dem Arbeitsspeicher auf die Diskette übertragen und anschließend sofort wieder von der Diskette in den Arbeitsspeicher kopiert.

Die beiden Wörter **DWTSECS** und **DRDSECS** übertragen stets ganzzahlige Vielfache von 256 Byte. Nun liegt nicht immer die Situation vor, daß wir genau 256 Byte oder ein Vielfaches davon speichern wollen. In diesem Fall bleibt uns jedoch nichts anderes übrig, als überschüssige Bytes mit abzuspeichern. In der Abbildung 8-2 werden die bisher besprochenen Wörter noch einmal vorgeführt. Dieses einfache Programm kann numerische Daten auf Diskette schreiben und anschließend wieder davon lesen. Wir vereinbaren für diesen Zweck zwei Arrays mit dem Namen **NUMB** und **NUMB1**. Das selbstdefinierte Wort **INPUT** beginnt in Zeile 2. Rufen wir es, so werden nacheinander die Zahlen 3, 6, 9, 12, ... , 597 im Array **NUMB** gespeichert. Wie Sie leicht überprüfen können, speichert der Array somit mehr als 256, aber weniger als 512 Byte. Das Wort für die Diskettenausgabe heißt **DISKWRITE** und beginnt in Zeile 4; es schreibt 512 Byte auf die Sektoren 8 und 9 von Spur 23 der Diskette im Laufwerk 1. Mit der Bytesübertragung wird dabei an der Adresse begonnen, die durch **NUMB** angegeben ist. Das Auftreten eines Schreibfehlers wird durch die Kombination aus **IF** und **THEN** in Zeile 5 abgefangen. In diesem Fall geben wir die Meldung "DISK ERROR" auf dem Bildschirm aus und brechen das Programm ab. Unser

```
0 ( Beispiel zur Disketten-Ein-/-Ausgabe )
 1 VARIABLE NUMB 512 ALLOT    VARIABLE NUMB1 512 ALLOT
 2 : INPUT      200  0  DO  I  3  *  NUMB  I  2  * +  !  LOOP  ;
 3
 4 : DISKWRITE NUMB  1  23  8  2  DWTSECS
 5             IF   ." DISK ERROR"  QUIT  THEN  ;
 6
 7 : DISKREAD  NUMB1  1  23  8  2  DRDSECS
 8             IF   ." DISK ERROR"  QUIT THEN   ;
 9
10 : OUTPUT    200  0  DO  NUMB1  2  I  *  + @   .   ."  " LOOP;
11
12
```

ABBILDUNG 8-2: Ein- und Ausgabe von Daten auf Diskette

Wort **INPUT** hat zwar nur 400 Byte des Arrays **NUMB** beschrieben; dennoch werden, beginnend bei **NUMB**, insgesamt 512 Byte auf Diskette übertragen. Die Anzahl der übertragenen Daten ist also unabhängig von der Größe des Arrays, mit dem wir gearbeitet haben.

In Zeile 7 beginnt die Definition von **DISKREAD**, welches 512 Byte von der Diskette in Laufwerk 1 liest und dabei die Sektoren 8 und 9 der Spur 23 heranzieht. Die übertragene Information wird im Arbeitsspeicher des Computers, beginnend bei der Adresse **NUMB1**, abgelegt. Wieder fangen wir einen eventuellen Ausgabefehler ab, wobei wir das Programm nach einer Fehlermeldung beenden. Jetzt fehlt noch ein Wort, mit dem wir die eingelesenen Daten anzeigen können; dies bewirkt die Definition von **OUTPUT**, die in Zeile 10 zu finden ist. Sie können somit überprüfen, ob die beiden Wörter **DISKWRITE** und **DISKREAD** richtig funktionieren.

Bei der Ausgabe der Daten auf Disketten haben wir einige Bytes mit übertragen, deren Inhalt uns gar nicht bekannt ist. Dies kommt daher, daß bei Disketten-Ein- und -Ausgabe immer nur 256 Byte oder ein ganzzahliges Vielfaches davon übertragen werden kann. In der Regel verursachen solche überschüssigen Bytes keine Probleme bei der Programmierung.

8.3 Eingabe von Textmaterial

Mit den bisher besprochenen Möglichkeiten kann ebensogut Textmaterial gespeichert werden; genaugenommen sind diese Verfahren auf keinen bestimmten Datentyp beschränkt. Wie wir jedoch erwähnten, sind **DWTSECS** und **DRDSECS** nicht Bestandteil von FORTH-79. Wir stellen deshalb an dieser Stelle noch einige FORTH-79-Befehle vor, mit denen Textmaterial aus einem Diskettenblock gelesen werden kann. Einige der Kommandos, die wir bereits aus Kapitel 7 kennen, können hier ebenfalls verwendet werden. So kennen wir z.B. aus Abbildung 7-4 bereits die Wörter **BLK**, **>IN** und **WORD**. Befindet sich vor Ausführung von **BLK** eine andere Zahl als 0 auf dem Stack, dann kommen die Daten nicht von der Tastatur, sondern von einem Diskettenblock. Die Nummer des Diskettenblockes entspricht dabei der Zahl, die **BLK** auf dem Stack vorfindet.

Es ist natürlich stets möglich, Textmaterial in einen Diskettenblock über den Editor einzugeben. Dies funktioniert genauso, wie wir Programme eingeben. Mit diesen Daten können wir dann die Verfahren anwenden, die wir in den letzten Abschnitten kennengelernt haben. Weiterhin stehen uns noch einige Wörter in FORTH-79 zur Verfügung.

TYPE - Mit diesem FORTH-Wort kann man eine bestimmte Anzahl Zeichen ausgeben. Wenn wir eingeben:

```
34500 60 TYPE
```

dann werden damit 60 Zeichen ausgegeben. Die ausgegebenen Zeichen befinden sich an den Adressen 34500 bis 34599 einschließlich. Beachten Sie, daß **TYPE** nicht unmittelbar mit Diskettenausgabe zu tun hat; vielmehr sollten wir es dann einsetzen, wenn wir Textmaterial auf den Bildschirm bekommen wollen, das in einem Blockpuffer gespeichert ist. Die Stack-Relation lautet:

a n --> (8-16)

COUNT - Ehe wir zeigen, wie man mit **TYPE** Textmaterial aus einem Blockpuffer ausgeben kann, wollen wir uns einem anderen FORTH-Wort, nämlich **COUNT**, zuwenden. Angenommen, in der Variablen **SENT** ist ein String gespeichert. Üblicherweise findet man in den er-

sten beiden Bytes dieser Variablen eine Integer, die die Länge des Strings angibt. Wir gehen davon aus, daß der String höchstens 64 Byte lang ist und zur Speicherung seiner Länge eine vorzeichenlose ein Bytes lange Zahl benötigt wird. Wenn wir nun **SENT** rufen, dann ist die auf den Stack gelegte Adresse nicht ausreichend, um von **TYPE** verarbeitet zu werden, da ja das erste Bytes eine Integer enthält. Diesem Umstand tragen wir Rechnung, indem wir die Zahl auf dem Stack einfach um eins erhöhen und anschließend noch eine Zahl auf den Stack pushen, die der Anzahl auszugebender Zeichen entspricht. Wir erhalten also das gewünschte Ergebnis mit folgenden Wörtern:

```
SENT 1+ SENT C@ TYPE                                   (8-17)
```

Dies sorgt dafür, daß sich vor dem Aufruf von **TYPE** sowohl die Anfangsadresse des Strings als auch seine Länge auf dem Stack befinden. Mittels **C@** legen wir die Zahl als einfach genaue Integer auf den Stack. Die einzelnen Verarbeitungsschritte von (8-17) kann man jedoch auch mit einem einzigen FORTH-Wort bewirken. Es lautet **COUNT**. Anstelle von (8-17) schreiben wir also einfach:

```
SENT COUNT TYPE                                        (8-18)
```

Eine eigene Definition von **COUNT** würde etwa folgendermaßen lauten:

```
:COUNT DUP 1+ SWAP C@ ;                                (8-19)
```

Wenden wir uns nun dem einfachen Programm in der Abbildung 8-3 zu, welches den Einsatz von **TYPE** für die Ausgabe von Textmaterial aus einem Speicherblock illustriert.

Das Beispiel bringt die Meldung DIESES PROGRAMM DRUCKT "DIESES PROGRAMM" auf den Bildschirm, an welche sich die Zeichenfolge DIESES PROGRAMM anschließt. In Zeile 1 sehen wir einen Teil des auszugebenden Texts in Klammern. Die Klammern sorgen dafür, daß beim Laden des Blockes die darin stehende Information weder com-

```
0 (Textausgabe mittels TYPE)
1 (DIESES PROGRAMM DRUCKT "DIESES PROGRAMM")
2 : TEST  CR CR   155  BLOCK   66 +   41   TYPE  CR CR
3          ." DIESES PROGRAMM"    CR CR    ;
4
5
6
```

ABBILDUNG 8-3: Ein einfaches Programmbeispiel zu TYPE

piliert noch ausgeführt wird. Wenn sich der Text auf einem separaten Block befindet, der nicht compiliert wird, dann wären die Klammern unnötig. Wir gehen davon aus, daß sich das Programm in Block 155 befindet. Wenn nun in Zeile 2 **BLOCK** gerufen wird, dann wird Block 155 in einen Speicherpuffer geladen und die Anfangsadresse des Puffers auf den Stack gelegt. Der Text, den wir ausgeben wollen, befindet sich in der ersten Zeile des Blockes. Jede Zeile eines Puffers enthält genau 64 Zeichen einschließlich Leerzeichen. Das Wort DIESES beginnt in der dritten Zeichenposition der zweiten Zeile. Somit ist der Buchstabe D das 67ste Zeichen im Puffer. Dies ist der Grund, weswegen wir die Anfangsadresse des Puffers, die wir über **BLOCK** erhalten, um 66 erhöhen. Der auszugebende Text ist insgesamt 41 Zeichen lang. Deshalb legen wir auch die Zahl 41 auf den Stack. Wird nun das Wort **TYPE** gerufen, dann liest es den gewünschten Text aus dem Puffer und gibt ihn aus. Den Rest der Meldung geben wir über das bereits bekannte FORTH-Kommando **."** aus. Mit den soeben eingeführten Methoden kann jeder auf Diskette gespeicherte Text ausgegeben werden.

8.4 Übungsaufgaben

Überprüfen Sie alle FORTH-Wörter oder Programme, die Sie in den folgenden Aufgaben schreiben, indem Sie sie auf Ihrem Computer laufen lassen. Halten Sie die einzelnen Wörter möglichst kurz. Programmieren Sie modular, indem Sie Teilaufgaben an Unterwörter delegieren.

8-1 Erörtern Sie die Prinzipien der Diskettenspeicherung.

8-2 Was ist ein Block?

8-3 Worin besteht der Unterschied zwischen einem Blockpuffer und einem Diskettenblock?

8-4 Schreiben Sie ein FORTH-Wort, das automatisch den gerade in Arbeit befindlichen Block lädt, wenn Sie den Befehl **LD** eingeben.

8-5 Probieren Sie die Wörter **EMPTY-BUFFERS, SAVE-BUFFERS, FLUSH** und **UPDATE** aus, um hinter ihre Funktionsweise zu kommen.

8-6 Worin besteht der Unterschied zwischen **EDIT** und **E**?

8-7 Was ist der Unterschied zwischen den Wörtern **LIST** und **L**?

8-8 Finden Sie heraus, wie **LOADS** in Ihrem System funktioniert.

8-9 Beschaffen Sie sich über das FORTH-Wort **INDEX** ein Listing der Programme, die auf Ihrer Diskette gespeichert sind. Dies setzt natürlich voraus, daß die erste Zeile eines Blockes die entsprechende Information enthält.

8-10 Ändern Sie das Programm der Abbildung 8-1 so ab, daß die Programme erst geladen werden, wenn der Benutzer das richtige Passwort eingibt.

8-11 Schreiben Sie ein FORTH-Wort, das ähnlich wie in Programm 8-1 Informationen über das Inhaltsverzeichnis einer Diskette liefert, wenn ihm für ein Programm folgende Informationen zur Verfügung stehen: sein Name, der Anfangsblock und die Anzahl der zu ladenden Blöcke.

8-12 Ändern Sie das Programm von Aufgabe 6-16 so, daß alle Ausgaben auf Diskette gehen.

8-13 Ändern Sie das Programm von Aufgabe 8-12 so, daß die Eingabedaten in einem Array gespeichert werden.

8-14 Schreiben Sie ein Programm, das die Fakultät der ersten fünf Zahlen berechnet und in einem Array speichert.

8-15 Wiederholen Sie Aufgabe 8-14 mit doppelt genauen Integers. Berechnen Sie die Fakultät der ersten neun Zahlen.

8-16 Wiederholen Sie Aufgabe 8-15 mit Gleitpunktzahlen.

8-17 Wiederholen Sie Aufgabe 8-15 mit doppelt genauen Gleitpunktzahlen.

8-18 Geben Sie mittels **COUNT** und **TYPE** einen String aus, der sich im Arbeitsspeicher befindet.

8-19 Wiederholen Sie Aufgabe 8-14, und drucken Sie diesmal mittels **TYPE** die passende Überschrift.

8-20 Speichern Sie Textmaterial auf einem Diskettenblock, und lassen Sie es dann satzweise ausgeben.

9

Einige weitere FORTH-Operationen

9 Einige weitere FORTH-Operationen

Dieses Kapitel bespricht einige zusätzliche FORTH-Wörter. Diese betreffen die Compilation und die Arbeitsweise des FORTH-Systems selbst. Kenntnisse dieser Wörter kommen Ihren Programmiertechniken und einem vertieften Verständnis von FORTH zugute.

9.1 Compilersteuerung

In diesem Abschnitt erörtern wir einige FORTH-Wörter, die die Arbeitsweise des Compilers beeinflussen. Bei allen bisherigen FORTH-Wörtern geschah folgendes: Während der Compilierung wird das Wort im Wörterbuch abgelegt und kann anschließend aufgerufen werden. Wenn ein Wort andere Wörter ruft, dann läuft im wesentlichen derselbe Vorgang ab. Nach der Compilierung enthält der Wörterbucheintrag des aufrufenden Wortes die Adresse des Wortes, das gerufen wird. Dadurch können "Unterwörter" als Teil des Abarbeitungsprozesses eines übergeordneten Wortes aufgerufen werden. Betrachten Sie dazu das Beispiel der Abbildung 7-5. Wir definieren darin die Wörter **ALPHA**, **ALPHAIN** und **ALPHAOUT**. Beim Laden des Blockes wird jedes dieser einzelnen Wörter compiliert und ins Wörterbuch eingetragen. Rufen wir nun das Wort **ALPHA**, so werden sowohl **ALPHAIN** als auch **ALPHAOUT** gerufen und ausgeführt. In FORTH gibt es jedoch einige Wörter, mit denen dieses allgemeine Schema abgeändert werden kann.

IMMEDIATE - Mit diesem FORTH-Wort kann man bewirken, daß ein anderes Wort während der Compilierung ausgeführt wird. Wie das geht, können Sie der Abbildung 9-1 entnehmen. Nehmen wir für den Augenblick einmal an, daß nur die Zeilen 1 und 2 in der Definition gegeben sind. Wenn wir den Block der Abbildung 9-1 laden, dann erscheint die Meldung "DRUCKE DAS" auf dem Bildschirm.

Diese Meldung erscheint, während der Block compiliert wird. Allerdings wird sie nicht durch die Compilation der ersten Zeile hervorgerufen. Der Compiler merkt sich lediglich beim Laden, daß für das Wort **TEST** der Status "IMMEDIATE" (deutsch "unmittelbar") vereinbart wurde. Als nächstes übersetzt er **TEST1** und trifft darin auf das Wort **TEST**; er weiß, daß es sich dabei um ein Wort mit dem Status "IMMEDIATE" handelt, weswegen nicht einfach die Adres-

```
0 ( Beispiel fuer Compilerworte )
1 : TEST  ." DRUCKE DAS " ;   IMMEDIATE
2 : TEST1 ." EINS "   TEST ;
3 : TEST2 ." ZWEI "        ;
4 : TEST3 ." DREI " [COMPILE] TEST      ;
5 : TEST4  ." VIER "  TEST  ;
6 : TEST5 ." FUENF "   COMPILE  TEST2   ;  IMMEDIATE
7 : TEST6 ." SECHS "  TEST5        ;
8
9
```

ABBILDUNG 9-1: Beispiele zur Compilersteuerung

se von **TEST** in die Definition von **TEST1** aufgenommen wird. Vielmehr wird das Wort **TEST** unmittelbar ausgeführt. Wenn wir jedoch nach vollständiger Compilation das Wort **TEST1** rufen, dann gelangt **TEST** nicht zur Ausführung. Ein Aufruf von **TEST1** bringt nicht die Meldung "DRUCKE DAS" auf den Bildschirm. Anders bei Eingeben von **TEST** (RETURN); in diesem Fall sehen wir die fragliche Meldung. Das bedeutet: Ist ein Wort mit dem Status "immediate" Teil der Definition eines anderen Wortes, dann wird dieses nicht ausgeführt, wenn das übergeordnete Wort gerufen wird. Wir verleihen einem Wort diesen Status, indem wir **IMMEDIATE** unmittelbar an den Strichpunkt der Definition anschließen.

[COMPILE] - Gelegentlich wollen wir ein Wort mit dem Status "immediate" in einem anderen FORTH-Wort verwenden, wobei es sich aber so verhalten soll, als wäre es ein gewöhnliches FORTH-Wort. Dies erreicht man mit dem Kommando **[COMPILE]**. Abbildung 9-1 gibt auch dafür ein Anwendungsbeispiel. Wir sehen, daß innerhalb der Definition von **TEST3** das Kommando **[COMPILE]** auftaucht. Es steht in diesem Fall unmittelbar vor dem FORTH-Wort, auf das es sich beziehen soll. Wenn nun das Wort **TEST3** der Abbildung 9-1 compiliert wird, dann verhält sich der Compiler so, als hätte **TEST** nicht den Status "immediate". Beim Laden des Blockes der Abbildung 9-1 verursacht also die Compilierung von **TEST3** nicht, daß die Meldung "DRUCKE DAS" erscheint. Das bedeutet also, daß ein Wort mit dem Status "immediate" nicht als ein solches behandelt wird, wenn ihm das Kommando **[COMPILE]** vorausgeht.

COMPILE - Wie wir jetzt wissen, wird ein Wort mit dem Status "immediate" ignoriert, wenn es in einem anderen Wort eingeschlossen ist und dieses compiliert wird. Das "immediate"-Wort wird in

den Wörterbucheintrag des umgebenden "äußeren" Wortes nicht mit aufgenommen. Sollte man dies jedoch wünschen, dann gibt es dafür einen speziellen Befehl, der dafür sorgt, daß "immediate"-Wörter auch in den Wörterbucheintrag anderer Wörter zur Compilezeit mit aufgenommen werden. Betrachten Sie dazu die Zeilen 3, 6 und 7 der Abbildung 9-1. Innerhalb der Definition von **TEST6** kommt das Wort **TEST5** vor, welches den Status "immediate" hat. Normalerweise würde bei einem Aufruf von **TEST6** das darin enthaltene **TEST5** keinerlei Auswirkungen auf den Programmablauf haben. In **TEST5** ist wiederum das Wort **TEST2** enthalten, bei dem es sich um ein normales Wort handelt. Unmittelbar vor dem Vorkommnis von **TEST2** innerhalb von **TEST5** finden wir nun das neue FORTH-Wort **COMPILE**. Wird nun **TEST5** innerhalb eines anderen Wortes compiliert, dann behandelt der Compiler **TEST2** so, als wäre es Teil eines gewöhnlichen Wortes und nicht eines mit dem Status "immediate". Die verbleibenden, in der Definition von **TEST5** noch enthaltenen Wörter werden jedoch so behandelt, wie es bei einem "immediate"-Wort üblich ist. Es wird also beispielsweise bei der Compilierung von **TEST6** die Meldung "FUENF" ausgegeben. Wird anschließend das Wort **TEST6** gerufen, dann sieht man hingegen die Meldung "SECHS ZWEI". Wäre nun innerhalb von **TEST5** das Wort **COMPILE** nicht enthalten, dann hätte **TEST2** keinerlei Auswirkung auf die Arbeitsweise von **TEST6**. Wird **TEST5** selbst ausgeführt, dann wird **TEST2** übergangen. Das bedeutet, daß das hinter **COMPILE** stehende Wort nicht ausgeführt wird, wenn das umschließende Wort gerufen wird. Rufen wir ein Wort, in dessen Definition der Befehl **COMPILE** enthalten ist, dann wird die Adresse des darauffolgenden Wortes mit in das Wörterbuch übernommen, so daß es ausgeführt werden kann.

Sehen wir uns noch einmal im einzelnen an, was beim Laden des Blockes der Abbildung 1-1 passiert. Die Compilierung von **TEST1** bewirkt, daß die Meldung "DRUCKE DAS" ausgegeben wird. Eben diese Meldung wird auch ausgegeben, wenn der Compiler **TEST4** übersetzt. Schließlich sehen wir noch die Meldung "FUENF", wenn der Compiler auf **TEST6** stößt.

Sehen wir uns nun an, was bei Ausführung der einzelnen Wörter in Abbildung 9-1 geschieht. Rufen wir **TEST**, dann erhalten wir die Meldung "DRUCKE DAS". Bei Ausführung von **TEST1** sehen wir auf dem Bildschirm die Meldung "EINS". Da **TEST** ein "immediate"-Wort ist, hat es hier keinerlei Auswirkungen. **TEST2** ist wieder ein ganz normales Wort, es bringt die Zeichenfolge "ZWEI" auf den Bildschirm. Rufen wir **TEST3**, so sehen wir "DREI DRUCKE DAS". Das Kommando **[COMPILE]** sorgt dafür, daß **TEST** mit in dieses Wort com-

piliert wurde. Das Wort in Zeile 5, **TEST4**, bringt nur die Meldung "VIER", da hier **TEST** unverändert seinen Status als "immediate" beibehält. Ebenfalls sehen wir bei Ausführung von **TEST5** lediglich die Meldung "FUENF". Hier sorgt **COMPILE** dafür, daß von einer Ausführung von **TEST2** abgesehen wird. Andererseits wird **TEST2** innerhalb von **TEST5** aktiv, wenn **TEST5** von einem anderen Wort gerufen wird. Diese Situation haben wir in **TEST6**, welches deswegen auch die Zeichenfolge "SECHS ZWEI" ausgibt.

LITERAL, [und] - Oft ist es nützlich, in einem Programm den Anfangswert einer Variablen längere Zeit zu benutzen. Nun kann sich jedoch der Variablenwert während der Ausführung ändern. Deshalb genügt es nicht, einfach den Variablenwert bei der Ausführung des Programms zu holen, weil sich dabei ja ein anderer als der Anfangswert ergeben kann. Eine Lösung wäre es, statt dessen eine Konstante zu definieren, die den Anfangswert der Variablen darstellt. Sollte diese Notwendigkeit aber bei mehreren Variablen auftreten, so würde dadurch zuviel Speicherplatz verschwendet werden. Mit den FORTH-Wörtern **LITERAL**, [und] kann man dieses Problem lösen. Eine Beispielanwendung dieser Wörter sehen Sie in Abbildung 9-2. Sehen wir uns einmal an, was hier geschieht. In Zeile 1 definieren wir die Variable **VAR**.

```
0 ( Beispielprogramm zu LITERAL )
1  VARIABLE  VAR          5   VAR !
2 : SUMME    [ VAR @ ]   LITERAL       +     ;
3 : ERGEBNIS  7    *    DUP    VAR !    SUMM    .   ;
4
5
```

ABBILDUNG 9-2: Ein Beispielprogramm zu **LITERAL**, [und]

Dann legen wir die Zahl 5 auf den Stack und speichern diesen Wert in **VAR**. In der nächsten Zeile sehen wir die Definition von **SUMME**, worin die FORTH-Wörter zum Holen des Variablenwerts in eckige Klammern eingeschlossen sind. Das Wort **Ä** sorgt dafür, daß anschließende Befehle ausgeführt und nicht compiliert werden, wenn der Block geladen wird. Die Ausführung geht so lange weiter, bis FORTH auf das Wort **Ü** stößt. Dahinter wird wieder normal mit der Compilierung weitergemacht. Das bedeutet, daß während der Compilierung von **SUMME** der Wert 5, der in **VAR** gespeichert ist, auf den Stack gelegt wird. Es folgt das Kommando **LITERAL**. Dieses sorgt

dafür, daß die einfach genaue Integer an oberster Stack-Position vom Stack entfernt und in die Compilierung von **SUMME** mit aufgenommen wird. Die gemeinsame Wirkung dieser Befehle ist also genauso, als hätten wir die Konstante 5 nach dem Wort **SUMME** in Zeile 2 geschrieben. Das selbstdefinierte **SUMME** entfernt also die oberste Zahl vom Stack und addiert den Wert 5 darauf. Das Ergebnis dieser Addition wird dann auf den Stack gelegt.

Betrachten Sie nun die Definition von **ERGEBNIS** in Zeile 3. Dieses Wort entfernt die oberste Zahl vom Stack, multipliziert sie mit 7 und dupliziert das Ergebnis. Das Produkt wird daraufhin in **VAR** gespeichert. Das Wort **ERGEBNIS** hat also als Ergebnis diejenige Zahl, die sich ergibt, wenn man den obersten Stack-Eintrag mit 7 multipliziert und darauf 5 addiert.

Wie wir sehen konnten, entfernt **LITERAL** die oberste einfach genaue Integer vom Stack und sorgt dafür, daß sie in den Wörterbucheintrag des Wortes eingeht, das gerade compiliert wird. Die Stack-Relation lautet also

n --> (9-1)

vorausgesetzt, die FORTH-Wörter [und] werden innerhalb einer Wortdefinition eingesetzt. Während des Compiliervorgangs werden Befehle, die zwischen eckigen Klammern stehen, ausgeführt und nicht mit übersetzt.

9.2 Alternative Wörterbücher

Wenn ein Wörtereintrag im Wörterbuch gespeichert wird, dann wird darin unter anderem die Adresse des nächsten Wörterbucheintrags mit vermerkt. Man spricht in diesem Fall von einem Kettungsfeld oder Zeiger. Jeder einzelne Wörterbucheintrag besteht seinerseits aus einer Folge von zusammenhängenden Speicherwörtern. Wenn das Wörterbuch durchsucht wird, dann beginnt man dabei von hinten, d.h., der letzte Wörterbucheintrag ist der erste, bei dem die Suche beginnt. Hat der letzte Wörterbucheintrag nicht den passenden Namen, dann geht man dem Zeiger im Verkettungsfeld nach und sucht mit dem vorletzten Eintrag weiter. Gelegentlich ist es

wünschenswert, in diesen Suchvorgang eine neue Ordnung einzuführen. Nehmen Sie etwa an, daß Sie ein Programm mit Ihrem Editor bearbeiten. Es wäre nun wünschenswert, daß FORTH zuerst nach den Editorkommandos sucht, da dadurch die Zeit für die Wörterbuchsuche verkürzt werden könnte. Bei anderen Anwendungen wäre es wiederum wünschenswert, andere Wortgruppen als erste durchsuchen zu lassen.

Wir wollen deshalb einmal sehen, wie der Aufbau des Wörterbuches und die Reihenfolge, in der es durchsucht wird, beeinflußt werden kann. Alle Einträge im Wörterbuch sind miteinander über das Kettungsfeld verknüpft. Wir können jedoch Teilmengen von Wörtern zusammengruppieren. Man spricht in FORTH im Zusammenhang mit einer solchen Gruppe von einem Vokabular. Das Wörterbuch braucht nicht eine lineare Abfolge verketteter Wörter zu sein; es sind auch Verzweigungen in dieser Struktur möglich. Nehmen wir z.B. an, in unserem Wörterbuch gibt es drei solche Verzweigungen oder parallele Stränge. In jedem Strang zeigt das Kettungsfeld eines Wortes zurück auf den Vorgänger in diesem Strang. Es ist jedoch ohne weiteres möglich, daß zwei Wörter aus zwei unterschiedlichen Strängen auf einen gemeinsamen Vorgänger in dem dritten Strang verweisen. Die ersten beiden Stränge verweisen allerdings nicht aufeinander. Jeden solchen Strang in einem Wörterbuch kann man nun als Vokabular auffassen. Das Hauptvokabular in der Programmiersprache FORTH trägt sinnigerweise den Namen **FORTH**. Alle bisher geschriebenen Definitionen wurden in dieses Vokabular eingetragen. Wie wir aber bereits gesagt haben, ist es gelegentlich wünschenswert, separate Unterwörterbücher (Vokabular) einzurichten. Wir wollen einmal sehen, wie dies erreicht werden kann.

VOCABULARY - Mit dem FORTH-Wort **VOCABULARY** richtet man ein neues Teilwörterbuch (ein neues Vokabular) ein. Wenn wir beispielsweise ein Vokabular mit dem Namen **HOUSE** vereinbaren wollen, dann geben wir zuerst das Kommando:

```
VOCABULARY HOUSE                                    (9-2)
```

In einigen FORTH-Versionen ist es nötig, hinter **HOUSE** das Kommando **IMMEDIATE** zu schreiben. Bis jetzt ist **HOUSE** noch kein Teilvokabular. Wenn wir jedoch das Wort **HOUSE** rufen, dann wird dieses zum neuen aktuellen Vokabular. Angenommen, das FORH-System durchsucht nun das Wörterbuch. Die Suche beginnt immer mit dem aktuel-

len Vokabular. Nur wenn darin das gesuchte Wort nicht enthalten ist, dann setzt das System die Suche mit dem FORTH-Vokabular fort. Wenn wir das Wort FORTH rufen, dann machen wir wieder dieses Vokabular zum aktuellen. Eine Wörterbuchsuche würde nun im FORTH-Wörterbuch beginnen. Findet sich jedoch nun das gesuchte Wort nicht, dann verzweigt die Suche in diesem Falle nicht zum Teilvokabular **HOUSE**. Indem wir die Definition von Teilwörterbüchern sorgsam planen, können wir damit sehr bequem arbeiten. Natürlich kann man genausogut mit ausschließlich einem Wörterbuch arbeiten. Viele Verarbeitungsschritte werden jedoch beschleunigt, wenn man sich alternativer Wörterbücher bedient.

CONTEXT, **CURRENT** und **DEFINITIONS** - Wir müssen unsere Ausführungen vom letzten Abschnitt dahingehend erweitern, daß ein Unterschied zwischen dem Wörterschatz besteht, der durchsucht und dem, an den neue Definitionen angefügt werden. Die Wörterbuchsuche findet im Kontextwörterbuch statt, während neue Definitionen ins aktuelle Wörterbuch aufgenommen werden. Wenn wir nun das FORTH-Wort **DEFINITIONS** rufen, dann wird das Kontext-Wörterbuch auch zum aktuellen. Wenn wir z.B. nach Ausführung von (9-2) zusätzlich noch **HOUSE** und anschließend **DEFINITIONS** ausführen, dann werden alle neuen Definitionen auch in das Teilwörterbuch **HOUSE** compiliert. Nocheinmal: Neue Einträge werden im aktuellen Wörterbuch vorgenommen, während Wörterbuchsuchen grundsätzlich im Kontextwörterbuch stattfinden.

FORTH merkt sich in zwei speziellen Variablen, welches Wörterbuch nun das Kontext- und welches das aktuelle Wörterbuch ist; sie heißen **CONTEXT** und **CURRENT**. In **CONTEXT** finden wir die Adresse des Wörterbuchnamens, bei dem die Wörterbuchsuche beginnen soll. Ähnlich steht in **CURRENT** die Adresse des Wörterbuches, in das neue Definitionen eingetragen werden sollen. Sowohl **CONTEXT** als auch **CURRENT** haben als Stack-Relation

--> a (9-3)

Beachten Sie, daß man durch einen Aufruf von **CONTEXT** bzw. **CURRENT** die Adresse dieser Variablen und nicht etwa die der entsprechenden Wörterbücher auf den Stack bekommt.

Wir wollen nun noch weitere Einzelheiten der Wörterbuchsuche besprechen. Angenommen, wir arbeiten mit drei verschiedenen Wörter-

büchern, die in der Reihenfolge vereinbart wurden: **FORTH, HOUSE** und **BOOK**. Falls **BOOK** das Kontextwörterbuch ist, dann bezieht eine Wörterbuchsuche alle drei Wörterbücher mit ein. Ist andererseits **HOUSE** das Kontextvokabular, dann wird **BOOK** bei der Suche übergangen.

' - Wir müssen jetzt noch das FORTH-Wort ' besprechen. Wird es ausgeführt, dann liegt die Adresse des nächsten Wortes im Eingabestrom auf dem Stack. Wenn wir z.B. folgende Wortfolge eingeben:

' NUMB (9-4)

dann erhalten wir als Ergebnis die Adresse von **NUMB** auf dem Stack. Dies setzt allerdings voraus, daß **NUMB** ein Name ist, der im Kontextwörterbuch bekannt ist. Sie können also überprüfen, ob das System einen bestimmten Namen momentan kennt, indem Sie ' und anschließend den gewünschten Namen eingeben. Erscheint die übliche Bereitschaftsmeldung Ihres Systems auf dem Bildschirm, dann bedeutet dies, daß der Name momentan bekannt ist. Sehen Sie statt dessen jedoch den Namen, gefolgt von einem Fragezeichen, dann heißt das, daß der fragliche Name durch eine Wörterbuchsuche nicht ausfindig gemacht werden kann. Ebenso kann man mit ' die Adresse einer Konstante finden und dann den darin gespeicherten Wert modifizieren.

9.3 Weitere FORTH-Kommandos

Die im Folgenden vorgestellten FORTH-Wörter befassen sich zum Teil mit Manipulationen des Arbeitsspeichers, zum Teil führen sie eine völlig neue Programmiertechnik ein.

MYSELF - Oft ist es wünschenswert, wenn sich ein FORTH-Wort selbst aufrufen kann; man spricht in diesem Fall von einer rekursiven Definition. Dazu setzt man das Kommando **MYSELF** ein. Wir wollen dies an einem Beispiel erläutern. In Abbildung 9-3 sehen Sie ein FORTH-Wort, dem wir den Namen **RECURSIVE** gegeben haben und das folgendermaßen funktioniert: Es legt eine 3 auf den Stack und führt dann die Addition aus. Die Summe wird dupliziert und anschließend um 99 vermindert. Schließlich überprüfen wir, ob wir

```
0 ( Ein rekursives Programm )
1 : RECURSIVE   3  +  DUP   99   -    0>
2         IF   . QUIT     THEN     MYSELF  ;
3
4
```

ABBILDUNG 9-3: Ein rekursives Programmbeispiel

einen Wert größer Null erhalten. Ist dies der Fall, dann wird der oberste Stack-Eintrag ausgegeben, und das Programm bricht ab. Ist die Zahl jedoch kleiner als 0, dann rufen wir das Wort **MYSELF**. Dies sorgt dafür, daß sich das Wort **RECURSIVE** selbst aufruft und den ganzen Vorgang wiederholt. Somit addiert unser Beispielprogramm fortgesetzt 3 auf eine Zahl, bis die entstehende Summe den Wert 100 übersteigt. In diesem Fall bricht der Prozeß ab, und die Ergebniszahl wird ausgegeben.

Beim Einsatz von **MYSELF** sollten Sie Vorsicht walten lassen. Achten Sie darauf, daß sich ein Wort nicht ununterbrochen selbst aufruft. In diesem Fall hängt sich das System nämlich auf, und Sie müssen Ihren Rechner neu starten. Auch sollten Sie vorsichtig sein, damit nicht fortgesetzt Daten auf den Stack gelegt werden und dieser somit den gesamten zur Verfügung stehenden Speicherplatz überschreibt.

EXECUTE - Wenn wir das Wort **EXECUTE** aufrufen, dann sorgt dieses dafür, daß das Wort, dessen Adresse auf dem Stack zu finden ist, ausgeführt wird.

EXIT - Wird **EXIT** innerhalb eines anderen Wortes gerufen, dann wird dessen Ausführung abgebrochen. Sie können **EXIT** jedoch nicht innerhalb von Schleifen verwenden. Befindet sich **EXIT** innerhalb eines Blockes und ist dort nicht Teil eines anderen Wortes, dann werden beim Laden dieses Blockes sowohl der Übersetzungs- als auch der Ladevorgang abgebrochen, wenn FORTH auf das Wort **EXIT** stößt.

FIND - Dieses Wort legt die Adresse des nächsten Wortes im Eingabestrom auf den Stack. Sollte es sich im Wörterbuch nicht ausfindig machen lassen, dann schreibt es statt dessen eine 0 auf den Stack.

Gelegentlich ist es wünschenswert, einen zusammenhängenden Block von Speicherwörtern mit einem bestimmten konstanten Wert zu belegen. Wir haben etwa in Abschnitt 8-2 erfahren, daß bei der Übertragung von Informationen auf Diskette eventuell überschüssige Bytes mit abgespeichert werden können. In diesem Fall wäre es wünschenswert, diese Überschußdaten zu löschen, so daß sich auf der Diskette keine sinnlosen Informationen befinden. Das Löschen könnte man z.B. dadurch erreichen, daß in den überschüssigen Speicherbereich an jede Adresse eine 0 geschrieben wird. Bei einigen Computern ist ein spezieller Bereich des Arbeitsspeichers für den Bildschirmspeicher reserviert. In diesem Fall ist der Computerbildschirm in einzelne Elemente unterteilt, von denen jedes die Größe eines Buchstabens oder Sonderzeichens hat. Jede dieser Bildschirmkomponenten entspricht nun genau einer Adresse im Bildschirmspeicher. Welches Zeichen an einer bestimmten Stelle des Bildschirms erscheint, hängt davon ab, welcher ASCII-Code sich an der Adresse befindet, die dieser Bildschirmstelle entspricht. Speicherworte, in denen der Bildschirminhalt abgebildet wird, sind stets zusammenhängend. Angenommen, wir wollen bei einem solchen Computersystem den Bildschirm löschen. Dazu müssen wir lediglich den ASCII-Code 32 für das Leerzeichen in jede Adresse des Bildschirmspeichers schreiben. Für die Erledigung solcher Aufgaben stellt FORTH mehrere Wörter zur Verfügung. Eines davon lautet **FILL**. Hier ein Anwendungsbeispiel:

```
26000 1000 90 FILL                                   (9-5)
```

Bei Ausführung von (9-5) werden 1000 Speicherwörter, beginnend bei der Adresse 26000, mit dem Zeichen besetzt, das den ASCII-Code 90 (der Buchstabe Z) hat. Die Stack-Relation für **FILL** lautet

$$a\ n_1\ n_2 \dashrightarrow \qquad (9\text{-}6)$$

Um einen Teil des Arbeitsspeichers mit Leerzeichen zu füllen, müssen wir nur für n_2 die Zahl 32 setzen. Es gibt jedoch noch ein anderes FORTH-Wort, das für diesen speziellen Zweck bequemer ist. Es lautet **BLANK** oder bei einigen Systemen **BLANKS**. Dieses Wort hat folgende Stack-Relation:

a n --> (9-7)

Hierbei werden n aufeinanderfolgende Speicherwörter mit Leerzeichen gefüllt, wobei bei der Adresse a begonnen wird.

ERASE - Dieses Wort funktioniert ähnlich wie **BLANK**, außer daß der Speicher nicht mit Leerzeichen, sondern mit binären Nullen besetzt wird.

9.4 Übungsaufgaben

Überprüfen Sie die Programme, die Sie in den folgenden Übungsaufgaben schreiben, indem Sie sie auf Ihrem Computer laufen lassen. Halten Sie einzelne Definitionen möglichst kurz, bedienen Sie sich des modularen Programmierstils.

9-1 Erörtern Sie die Funktionsweise von **IMMEDIATE**.

9-2 Legen Sie die Unterschiede zwischen [**COMPILE**] und **COMPILE** dar. Schreiben Sie dazu eigene FORTH-Wörter, die den Unterschied erhellen.

9-3 Schreiben Sie ein Programm, das jedesmal dann automatisch ein Disketteninhaltsverzeichnis ausgibt, wenn der Diskettenblock mit dem Inhaltsverzeichnis geladen wird.

9-4 Ändern Sie das Programm in Abbildung 7-2 so ab, daß bei seiner Übersetzung jedesmal ein Punkt ausgegeben wird, wenn eine Textzeile verarbeitet wird.

9-5 Erörtern Sie die Einsatzmöglichkeiten für das FORTH-Wort **LITERAL**.

9-6 Wiederholen Sie Aufgabe 9-5 mit den FORTH-Wörtern [und].

9-7 Schreiben Sie ein FORTH-Programm, das den Durchschnittswert eines Studenten in vier Tests ermittelt und das Ergebnis in einer Variablen speichert. Das Programm soll den Durchschnitt sowie eine Meldung ausgeben, ob der Student bestanden hat. Die Punktzahl zum Bestehen der Prüfung soll am An-

fang in derselben Variablen gespeichert werden, in der sich letztendlich der Durchschnitt befindet.

9-8 Schreiben Sie ein rekursives FORTH-Programm, das die Summe von 20 aufeinanderfolgenden Zahlen berechnet, wobei es mit der Zahl beginnt, die sich an oberster Stack-Position befindet.

9-9 Wiederholen Sie Aufgabe 9-8, wobei diesmal jedoch 20 aufeinanderfolgende gerade Zahlen addiert werden sollen.

9-10 Wiederholen Sie Aufgabe 9-8, wobei diesmal 20 aufeinanderfolgende ungerade Zahlen addiert werden sollen.

9-11 Schreiben Sie unter Verwendung von **MYSELF** ein rekursives Programm zur Berechnung der Fakultät.

9-12 Schreiben Sie ein rekursives Programm, das die Summe des Ausdrucks $1/n^2$ für n = 1, 2, 3, ... berechnet. Arbeiten Sie mit Gleitkommazahlen. Das Programm sollte abbrechen, wenn $1/n^2$ kleiner als 0.00001 wird.

9-13 Füllen Sie 1024 zusammenhängende Speicheradressen mit dem ASCII-Wert für X.

9-14 Warum kann das Programm der letzten Aufgabe unter Umständen dazu führen, daß Sie Ihr System neu starten müssen?

9-15 Schreiben Sie ein Programm, das einen Blockpuffer mit Leerzeichen auffüllt.

Anhang

Anhang: Glossar der FORTH-Wörter

Dieses Glossar führt alle besprochenen FORTH-Wörter auf. Sie finden darin den Abschnitt, in dem das entsprechende Wort besprochen wird, seine Stack-Relation und eine kurze Beschreibung der Funktionsweise. Nicht alle im Glossar enthaltenen Wörter sind Bestandteil von FORTH-79 Darüber hinausgehende Befehle sind in der Regel in MMSFORTH und anderen FORTH-Systemen zu finden. Genauere Informationen entnehmen Sie bitte den Abschnitten, in denen die einzelnen Wörter definiert werden.

BESCHREIBUNG	**ABSCHN.**	**STACKRELATION**	**WORT**
Speichert eine Zahl in der Adresse an oberster Stack-Position	6-2	n a -->	**!**
Dient bei der Zahlenausgabe mit Maske für die Zifferndarstellung vorzeichenloser doppelt genauer Integers	3-3	d_1 --> d_2	**#**
Beendet die maskierte Zahlenausgabe	3-3	d --> a n	**#>**
Fordert zur Eingabe einer einfach genauen Integer auf	3-1	--> n	**#IN**
Wandelt bei der Zahlenausgabe mit Maske Ziffernzeichen in den ASCII-Code um	3-3	d --> 0	**#S**
Dient zum Speichern von Strings	7-3	$a_\$$ a -->	**$!**
Vereinbart einen String im Arbeitsspeicher	7-3	--> a	**$"**
Entfernt nachlaufende Blanks vom String	7-3		**$-TB**
Druckt einen String	7-3	a -->	**$.**

Vereinbart einen String-Array	7-3	n_1 n_2 -->	**$ARRAY**
Vergleicht Stringvariable	7-3	a_1 a_2 --> n	**$COMPARE**
Vereinbart eine Stringkonstante	7-3		**$CONSTANT**
Vereinbart eine Stringvariable	7-3		**$VARIABLE**
Vertauscht die Werte in Stringvariablen	7-3	a_1 a_2 -->	**$XCG**
Liefert die Adresse des nächsten Wortes im Eingabestrom	9-2	a	**'**
Leitet einen Kommentar ein	2-3		**(**
Liefert das Produkt zweier Zahlen	2-1	n_1 n_2 -->	*****
Multipliziert n_1 mit n_2 und dividiert das doppelt genaue Produkt durch n_3	5-2	n_1 n_2 n_3 --> n	***/**
Ähnlich wie */; liefert jedoch auch den Rest	5-2	n_1 n_2 n_3 --> n_r n_q	***/MOD**
Liefert die Summe zweier Zahlen	2-1	n_1 n_2 --> n	**+**
Inkrementiert den gespeicherten Wert	6-2	n a -->	**+!**
Inkrementiert eine Schleifenvariable	4-3	n -->	**+LOOP**
Compiliert n ins Wörterbuch	6-3	n -->	**,**
Subtrahiert n_2 von n_1	2-1	n_1 n_2 -->	**-**
Aktualisiert den Zeichenzähler	7-3	a n_1 --> a n_2	**-TRAILING**
Gibt eine Zahl aus	1-4	n -->	**.**
Gibt Text aus	3-1		**."**

Gibt die Zahl n_1 im Datenfeld n_2 aus	3-3	n_1 n_2 -->	**.R**
Dividiert n_1 durch n_2	2-1	n_1 n_2 --> q	**/**
Division mit Quotient und Rest	2-1	n_1 n_2 --> r q	**/MOD**
"Wahr", falls $n < 0$	4-1	n --> f	**0<**
"Wahr", falls $n = 0$	4-1	n --> f	**0=**
"Wahr", falls $n > 0$	4-1	n --> f	**0>**
Inkrementiert den obersten Stack-Eintrag um Eins	2-7	n --> n_1	**1+**
Dekrementiert den obersten Stack-eintrag um Eins	2-7	n --> n_1	**1-**
Multipliziert den obersten Stack-Eintrag mit 16	2-7	n --> n_1	**16***
Speichert eine doppelt genaue Integer	6-2	d a -->	**2!**
Definiert einen zweidimensionalen String-Array	7-3	n_1 n_2 n_3 -->	**2$ARRAY**
Multipliziert die oberste Integer mit 2	2-7	n --> n_1	**2***
Addiert 2 auf die oberste Integer	2-7	n --> n_1	**2+**
Subtrahiert 2 von der obersten Integer	2-7	n --> n_1	**2-**
Dividiert die oberste Integer durch 2	2-7	n --> n_1	**2/**
Holt eine doppelt genaue Integer	6-2	a --> d	**2@**
Definiert einen zweidimensionalen Array	6-4	n_1 n_2 -->	**2ARRAY**

Beschreibung	Kap.	Stack	Wort
Definiert eine doppelt genaue Konstante	6-1	d -->	**2CONSTANT**
Definiert einen doppelt genauen Integer-Array	6-4	n_1 n_2 -->	**2DARRAY**
Entfernt die oberste doppelt genaue Integer vom Stack	2-5	d -->	**2DROP**
Dupliziert die oberste doppelt genaue Integer auf den Stack	2-5	d --> d d	**2DUP**
Dupliziert die zweite doppelt genaue Integer auf dem Stack an oberste Stack-Position	2-5	d_1 d_2 --> d_1 d_2 d_1	**OVER**
Rotiert die dritte doppelt genaue Integer an oberste Stack-Position	2-5	d_1 d_2 d_3 --> d_2 d_3 d_1	**2ROT**
Vertauscht die obersten beiden doppelt genauen Integers	2-5	d_1 d_2 --> d_2 d_1	**2SWAP**
Vereinbart eine doppelt genaue Variable	6-2		**2VARIABLE**
Leitet die Definition eines FORTH-Wortes ein	2-3		**:**
Beendet die Definition eines FORTH-Wortes	2-3		**;**
Wird "wahr", falls $n_1 < n_2$	4-1	n_1 n_2 --> f	**<**
Leitet die Zahleneingabe mit Maske ein	3-3		**<#**
Wird "wahr", falls n_1 kleiner oder gleich n_2 ist	4-1	n_1 n_2 --> f	**<=**
Wird "wahr", falls n_1 ungleich n_2 ist	4-1	n_1 n_2 --> f	**<>**
Dupliziert n Speicherwörter beginnend bei a_1 an der Adresse a_2;	7-1	a_1 a_2 n -->	**<CMOVE**

Übertragung beginnt bei der höchstwertigen Adresse			
Ist "wahr", falls n_1 gleich n_2 ist	4-1	n_1 n_2 --> f	**=**
Ist "wahr", falls n_1 größer oder gleich n_2 ist	4-1	n_1 n_2 --> f	**>=**
Enthält die Startposition für die Untersuchung des Eingabestroms	7-2	--> a	**<IN**
Überträgt eine Integer auf den Kontroll-Stack; benötigt entsprechendes R>	2-6	n -->	**>R**
Dupliziert die oberste einfach genaue Integer, es sei denn, diese ist gleich 0	4-5	n --> n n	**?DUP**
Holt die an der Adresse gespeicherte einfach genaue Integer	6-2	a --> n	**@**
Ersetzt die oberste einfach genaue Integer durch ihren Absolutbetrag	2-7	n_1 --> n_2	**ABS**
Erweitert den Speicherbereich einer Variablen um n Byte	6-3	n -->	**ALLOT**
Bitweises logisches AND	5-5	n_1 n_2 --> n_3	**AND**
Vereinbart einen Array			**ARRAY**
Legt den ASCII-Wert des ersten Zeichens in dem String, der bei a beginnt, auf den Stack	6-3	n -->	**ASC**
Enthält die Ein-/Ausgaberadix	6-3	a --> n	**BASE**
Leitet eine Schleife ein	2-7	--> a	**BEGIN**
Füllt Speicherbereiche mit Leerzeichen	9-3	a n -->	**BLANK**

Enthält die Adresse des Blockpuffers für den Eingabestrom	7-2	--> a	**BLK**
Überträgt den Block n von der Diskette in den Arbeitsspeicher und legt dessen Startadresse auf den Stack	8-1	n --> a	**BLOCK**
Wie BLOCK, die Daten werden jedoch nicht übertragen	8-1	n --> a	**BUFFER**
Speichert das niedrigstwertige Byte einer einfach genauen Integer	7-1	n a -->	**C!**
Holt ein Byte und speichert es als einfach genaue Integer	7-1	a --> n	**C@**
Beendet eine CASE-Anweisung	4-6		**CASEND**
Wandelt eine ein Byte lange Integer in ihre ASCII-Darstellung um; das Ergebnis steht im temporären Arbeitsbereich, dessen Adresse auf den Stack gelegt wird	7-3	c --> a	**CHR$**
Überträgt n Bytes von Adresse 1 nach Adresse 2; die Übertragung beginnt bei den niedrigwertigen Adressen	7-1	a_1 a_2 n -->	**CMOVE**
Nimmt einen Wert in die Wortdefinition mit auf	9-1		**COMPILE**
Vereinbart eine Konstante mit dem Wert n	6-1	n -->	**CONSTANT**
Enthält die Adresse des Kontext-Vokabulars	9-2	--> a	**CONTEXT**
Legt die Anfangsadresse des Strings und den Stringzähler auf den Stack	8-2	a --> a_1 n	**COUNT**

Beschreibung	Kap.	Stack	Wort
Sendet einen Zeilenvorschub	3-1		**CR**
Richtet einen Wörterbucheintrag ein	6-3		**CREATE**
Lenkt die Ausgabe auf den Bildschirm	3-1		**CRT**
Enthält die Adresse des aktuellen Wörterbuches	9-2	--> a	**CURRENT**
Fordert zur Eingabe einer doppelt genauen Integer auf	5-1	--> d	**D#IN**
Multipliziert doppelt genaue Integers	5-1	d_1 d_2 --> d_p	**D***
Multipliziert d_1 mit d_2 und dividiert das vierfach genaue Produkt anschließend durch d_3	5-2	d_1 d_2 d_3 --> d	**D*/**
Wie D*/; liefert aber auch den Rest	5-2	d_1 d_2 d_3 --> d_r d_q	**D*/MOD**
Addiert zwei doppelt genaue Zahlen	5-1	d_1 d_2 --> d	**D+**
Subtrahiert zwei doppelt genaue Zahlen (d_1 minus d_2)	5-1	d_1 d_2 --> d	**D-**
Liefert den Quotienten von d_1 und d_2	5-1	d_1 d_2 --> d	**D/**
Wie D/, liefert aber auch noch den Rest	5-1	d_1 d_2 --> d_r d_q	**D/MOD**
Gibt eine doppelt genaue Integer aus	5-1	d -->	**D.**
Gibt eine doppelt genaue Integer in einem n Zeichen langen Datenfeld aus	5-1	d n -->	**D.R**

Beschreibung	Kap.	Stack	Wort
Ist "wahr", wenn der doppelt genaue Wert gleich 0 ist	5-1	d --> f	**D0=**
Ist "wahr", wenn d_1 kleiner d_2 ist	5-1	d_1 d_2 --> f	**D<**
Liefert den Absolutwert einer doppelt genauen Integer	5-1	d_1 --> d_2	**DABS**
Vereinbart einen Array mit doppelt genauen Integers	6-3	n -->	**DARRAY**
Setzt die Zahlenbasis auf 10	2-7		**DECIMAL**
Macht den Kontext-Wortschatz zum aktuellen Wortschatz	9-2		**DEFINITIONS**
Liefert die Stack-Tiefe in Einheiten von einfach genauen Integers	2-2	--> n	**DEPTH**
Liefert die größere von zwei doppelt genauen Integers	5-1	d_1 d_2 --> d	**DMAX**
Liefert die kleinere von zwei doppelt genauen Integers	5-1	d_1 d_2 --> d	**DMIN**
Dreht das Vorzeichen einer doppelt genauen Integer um	5-1	d --> -d	**DNEGATE**
Leitet eine Schleife ein	4-3	n_1 n_2 -->	**DO**
Liest Diskettensektoren	8-2	a n_1 n_2 n_3 n_4 --> nf	**DRDSECS**
Entfernt die oberste einfach genaue Integer vom Stack	2-2	n -->	**DROP**
Dupliziert die oberste einfach genaue Integer	2-2	n --> n n	**DUP**
Schreibt Diskettensektoren	8-2	a n_1 n_2 n_3 n_4 --> nf	**DWTSECS**

Bearbeitet den Block, der vom Inhalt von SCR bestimmt wird	8-1		**E**
Bearbeitet Block n; n wird in SCR gespeichert	4-2	n -->	**EDIT**
Für Programmverzweigungen	4-2		**ELSE**
Gibt ein Zeichen aus	7-1	c -->	**EMIT**
Markiert alle Puffer als leer	8-1		**EMPTY-BUFFERS**
Setzt n aufeinanderfolgende Byte auf den Wert 0, beginnend mit der Adresse a	9-3	a n -->	**ERASE**
Führt den Wörterbucheintrag aus, dessen Adresse auf dem Stack liegt	9-3	a -->	**EXECUTE**
Beendet die Programmbearbeitung	9-3		**EXIT**
Liest Zeichen in den Arbeitsspeicher ein, beginnend bei Adresse a, wobei maximal n Zeichen oder bis zum ersten Return gelesen wird	7-2	a n -->	**EXPECT**
Belegt n aufeinanderfolgende Speicherwörter (beginnend bei Adresse a) mit dem ASCII-Wert n_c	9-3	a n n_c -->	**FILL**
Sucht die Adresse des nächsten Wortes im Eingabestrom	9-3	--> a	**FIND**
Speichert die markierten Puffer auf Diskette	8-1		**FLUSH**
Löscht alle Wörter bis einschließlich dem angegebenen aus dem Wörterbuch	2-4		**FORGET**
Name des Hauptwörterbuches	9-2		**FORTH**

Beschreibung	Kapitel	Stack	Wort
Liefert die Adresse des nächsten verfügbaren Wörterbuchbytes	7-2	--> a	**HERE**
Umwandlung der Zahlenausgabe in Hexadezimaldarstellung	2-7		**HEX**
Zur Einfügung von Zeichen bei der Zahlenausgabe mit Maske	3-3	c -->	**HOLD**
Legt den Schleifenindex auf den Stack	4-3	--> n	I
Legt den Testwert der Schleife auf den Stack	4-3	--> n	I'
Für Programmverzweigungen	4-2	f -->	**IF**
Schaltet von Compilierung in Ausführung um	9-1	n_1 n_2 -->	**IMMEDIATE**
Gibt die erste Zeile von n_2 Blockes aus, beginnend mit Block n_1	9-1	n_1 n_2 -->	**INDEX**
Liefert den Index der dynamisch übernächsten Schleife auf den Stack	4-3	--> n	J
Legt den ASCII-Code des nächsten Eingabezeichens auf den Stack	7-1	--> n	**KEY**
Gibt den Block aus, dessen Nummer in SCR gespeichert ist	8-1		L
Beendet eine Schleife	4-3		**LEAVE**
Überträgt die n ersten Zeichen des Strings, der bei a beginnt, in den temporären Arbeitsbereich	7-3	a n --> a_1	**LEFT$**
Legt die Länge eines Strings auf den Stack	7-3	a --> n	**LEN**

Beschreibung		Stack	Wort
Gibt den Block n aus und legt n in SCR ab	2-3	n -->	**LIST**
Nimmt den Stack-Wert, ohne ihn zu interpretieren, in die Compilation mit auf	9-1	n -->	**LITERAL**
Lädt den Block n	2-3	n -->	**LOAD**
Lädt n_2 Block, beginnend mit n_1	8-1	n_1 n_2 -->	**LOADS**
Inkrementiert den Schleifenindex	4-3		**LOOP**
Doppelt genaues Produkt zweier einfach genauer Integers	5-3	n_1 n_2 --> d	**M***
Multipliziert d_1 mit n_2 und speichert das Produkt als dreifach genaue Integer, welche dann durch n_3 dividiert wird; der Quotient ist doppelt genau	5-3	d_1 n_2 n_3 --> d_2	**M*/**
Gemischte Addition	5-3	d_1 n --> d_2	**M+**
Gemischte Subtraktion	5-3	d_1 n --> d_2	**M-**
Gemischte Division	5-3	d n_1 --> n_2	**M/**
Wie M/, außer daß sowohl Quotient als auch Rest geliefert werden	5-3	d n_1 --> d_r n_q	**M/MOD**
Liefert den größeren von zwei Werten	2-7	n_1 n_2 --> n	**MAX**
Überträgt an die Adresse a_1 einen n_2 Zeichen langen Teilstring, der ab der $n_1$ten Zeichenposition des Strings a beginnt	7-3	a n_1 n_2 --> a_1	**MID$**
Liefert den kleineren von zwei Werten	2-7	n_1 n_2 --> n	**MIN**
Liefert den Rest der Division von n_1/n_2	2-1	n_1 n_2 --> n	**MOD**

Beschreibung	Kapitel	Stack	Wort
Verschiebt n 16 Byte lange Speicherwörter, beginnend bei a_1, nach n_2	7-1	a_1 a_2 n -->	**MOVE**
Erlaubt rekursive Aufrufe	9-3		**MYSELF**
Leitet eine CASE-Anweisung ein	4-6	n -->	**NCASE**
Ersetzt eine Zahl durch die negative Zahl mit dem gleichen Betrag	2-7	n --> -n	**NEGATE**
Negiert ein Flag	4-5	f_1 --> f_2	**NOT**
Setzt die Ein-/Ausgabebasis für Zahlen auf das Oktalsystem	2-7		**OCTAL**
Allgemeiner Ausgang in CASE-Anweisungen	4-6		**OTHERWISE**
Dupliziert die zweite Zahl an oberste Stack-Position	2-2	n_1 n_2 --> n_1 n_2 n_1	**OVER**
Enthält die Anfangsadresse des temporären Arbeitsbereichs	7-3	--> a	**PAD**
Löscht den Bildschirm	3-1		**PAGE**
Legt die Ausgabe sowohl auf Bildschirm als auch auf Drucker	3-1		**PCRT**
Legt die Ausgabe nur auf den Drucker	3-1		**PRINT**
Für Zeicheneingabe	7-2		**QUERY**
Löscht den Return-Stack	3-1		**QUIT**
Überträgt die oberste einfach genaue Integer vom Return-Stack auf den Parameter-Stack	2-6	--> n	**R>**
Dupliziert die oberste einfach genaue Integer vom Return-Stack auf den Parameter-Stack	2-6	--> n	**R @**

Initialisiert den Zufallszahlengenerator	2-7		**RANDOMIZE**
Für die Programmierung von Schleifen	4-4		**REPEAT**
Überträgt die n letzten Zeichen des Strings a in den temporären Arbeitsbereich, liefert dessen Adresse	7-3	a n --> a_1	**RIGHT$**
Erzeugt eine Zufallszahl und speichert sie in SEED	2-7		**RN1**
Erzeugt eine Zufallszahl zwischen 1 und n_1	2-7	n_1 --> n_2	**RND**
Legt die n-te einfach genaue Integer auf dem Stack an oberste Stack-Position	2-2	n -->	**ROLL**
Befördert die dritte einfach genaue Integer an oberste Stack-Position	2-2	n_1 n_2 n_3 --> n_2 n_3 n_1	**ROT**
Markiert alle Puffer für nachfolgende Sicherungen	8-1		**SAVE-BUFFERS**
Enthält die Adresse des zuletzt bearbeiteten Blockpuffers	8-1	--> a	**SCR**
Fügt den ASCII-Code des Minuszeichens bei Zahlenausgabe mit Maske ein, falls n negativ ist	3-3	n -->	**SIGN**
Gibt ein Leerzeichen aus	3-1		**SPACE**
Vertauscht die beiden obersten Stack-Einträge	2-2	n_1 n_2 --> n_2 n_1	**SWAP**
Bei Programmverzweigungen benötigt	4-2		**THEN**

Beschreibung	Kapitel	Stapel	Wort
Gibt n Zeichen beginnend ab der Adresse a aus	3-3	a n -->	**TYPE**
Vorzeichenlose Integermultiplikation	5-4	u_1 u_2 --> u	U*
Gibt eine vorzeichenlose Integer aus	5-4	u -->	U.
Gibt eine vorzeichenlose Integer in einem n Stellen breiten Datenfeld aus	5-4	u n -->	U.R
Vorzeichenlose Division mit doppelt genauem Dividenden, liefert Quotienten und Rest	5-4	u_d u_1 --> u_r u_q	**U/MOD**
"Wahr", falls u_1 kleiner u_2 ist (vorzeichenlose Integers)	5-4	u_1 u_2 -->	U<
Für die Programmierung von Schleifen	4-4	f -->	**UNTIL**
Markiert alle Blockpuffer als gesichert	8-1		**UPDATE**
Wandelt die ASCII-Darstellung einer Integer in Binärform um	7-3	a --> n	**VAL**
Definiert eine Variable	6-2		**VARIABLE**
Für die Vereinbarung eines neuen Wortschatzes	9-2		**VOCABULARY**
Für die Programmierung von Schleifen	4-4	f -->	**WHILE**
Liest Zeichen aus dem Eingabestrom; Trenner ist Zeichen mit ASCII-Code n	7-2	n --> a	**WORD**
Bitweises exklusives ODER	5-5	n_1 n_2 --> n	**XOR**

Fragt nach Y oder N; N liefert Wahrheitswert "wahr"	4-5 --> f	**Y/N**
Beendet Compilierung und leitet Ausführung ein; wird in Wortdefinitionen benötigt	9-1	**[**
Bewirkt, daß ein Wort mit dem Status IMMEDIATE compiliert wird	9-1	**[COMPILE]**
Beendet Ausführung und fährt mit der Compilierung fort	9-1	**]**

Stichwortverzeichnis

Stichwortverzeichnis

T

U

V

Der Einstieg in FORTH

FORTH — die Sprache für alle, die mehr aus ihrem Computer herausholen wollen!
Wer einen Computer hat, braucht nur noch dieses Buch, um in kurzer Zeit eigene Programme in dieser vielseitigen Sprache schreiben zu können. Dafür sorgt die gründliche und methodische Methode des Autors, der an einer Vielzahl genau erläuterter Beispiele alles Wichtige erklärt. Der Leser findet jedoch nicht nur eine Einführung in das Programmieren mit FORTH; viele andere wichtige Themen werden erörtert, unter anderem

- Vergleich von FORTH mit anderen Sprachen
- Editieren von Programmen
- Fehlersuche und -korrektur
- Diskettenoperationen
- Zahlentypen
- Grundlagen des strukturierten Programmierens
- Der FORTH-Standard FORTH-79 und Erweiterungen
- Ausführliches Glossar

Markt&Technik Verlag Aktiengesellschaft
ISBN 3-89090-085-2

DI
Sfr. 53,40